UNIVERSITY OF OSLO
Department of Informatics

Organizing objects in a high density storage of stacks

Master thesis

Stian Lind Petlund

May 4, 2015

Organizing objects in a high density storage of stacks

Stian Lind Petlund

May 4, 2015

Abstract

Today, industrial and service robots are used all over the world to handle simple repetitive tasks as well as more complex tasks. Many warehouses use robots for their logistical tasks, while warehouse employees only need to ensure that everything is performing properly and finish some tasks like collecting items and shipping them out to customers.

This thesis investigates how to reorganize a high density storage of stacks. Multiple algorithms such as exhaustive, random and greedy searches were implemented on top of my own problem specific designs, and tested in simulations separated from the actual system. The results from implementations made during the work with this thesis were compared to each other and to the performance of today's system.

The majority of the work in this thesis have been to design and implement my own ideas on how to solve the reorganization problem. During my work, much time have been spent to rethink, adjust and redesign the algorithms in order to get even better performance. Results from simulations of the various algorithms are promising, and improvements can be seen over the different implementations during my work.

In comparison with how today's system performs on the reorganization task, the results are also very promising. However, the comparison of today's system and the implementations in this thesis have some uncertainties, and further work is required to confirm the improvement completely. The last chapter of this thesis contain useful suggestions on further work may improve the reorganization time even more.

Keywords. Multiple stacks, combinatorial optimization, logistics, storage.

Preface and acknowledgements

This thesis has been submitted to the Department of Informatics, The Faculty of Mathematics and Natural Sciences at University of Oslo (UiO) spring 2015. The research has been done at UiO with valuable support from the company that invented and operates the target storage.

I would like to thank my supervisor Professor Jim Tørresen for all the valuable advice and support during my work with this thesis. I would also like to thank Ragnar Stuhaug who is the domain expert of the target system and helped me focus on the important parts of the system.

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

With today's challenges in planning and organizing complex tasks, the applications of combinatorial optimization are numerous. Small and large industries face challenges of great diversity, and the algorithms dealing with these problems are often optimized for specific problems. When problems of a certain complexity grow too large, they become intractable. At this point and an exhaustive search for the optimal solution will not be feasible.

Various methodologies such as randomized algorithms, greedy search and evolutionary algorithms can be used to solve intractable problems. These algorithms may not find the optimal solution, nor provide a proof that the optimal solution is found, but they are often highly efficient and can give very good suboptimal solutions in short time.

In this thesis, I will investigate a combinatorial optimization problem concerning logistics in a high density storage. The storage is a grid of stacks, where the stacks are placed firmly against each other. Only the top object in a stack can be moved from one stack to another.

1.2 Goal of the thesis

As different objects are retrieved and delivered to warehouse employees, the grid gets unordered over time. The goal of this thesis is to investigate different methods to reorganize the objects in the grid such that higher prioritized objects are easily accessible on the surface, while non-prioritized objects are stored deeper in each stack.

1.3 Research method

My research is based on experiments [1] with different algorithms I have designed and implemented. I will use a hypothetico-deductive method, which has the following pattern:

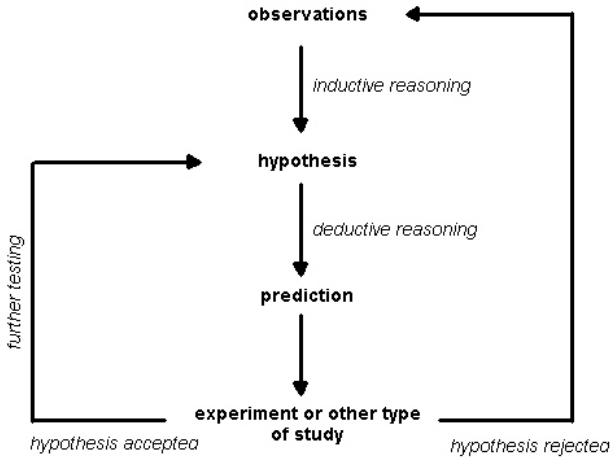

Figure 1.1: A flowchart for the hypothetico-deductive method

Considering a problem or subproblem (observation), I will provide a suggested, testable solution (hypothesis) to solve the problem. From the hypothesis, predictions can be deduced and an experiment will be performed to check if the hypothesis has validity.

Because the experiments performed are influenced by a various degree of randomness, no experiment will be able to fully verify a hypothesis. However, by carrying out multiple repeated experiments, positive results from experiments will provide stronger confirmation that a certain hypothesis is correct.

1.4 Outline

This thesis has five chapters: Introduction, Background, Methodologies and Experiments, Results and Conclusion.

Chapter 2, Background, gives the reader an introduction to different warehouse robots, and presents the grid in necessary detail. A background on combinatorial optimization can be found in the end of the chapter.

Chapter 3, Methodologies and experiments, starts with a problem definition of the reorganization problem to be solved. Then different algorithms are suggested,

among them are exhaustive search, random algorithms, iterative approaches and greedy search.

Chapter 4, Results, contains results on how the implementations of different methods discussed in chapter 3 works. This chapter will also provide some further discussion as we get familiar with how the different algorithms perform. Sometimes, minor modifications are suggested, implemented and compared.

Chapter 5 Conclusion and further work, contains a conclusion on whether the initial goal of the thesis was reached or not. Because a lot of potential methodologies could not be tested and reported in the results chapter, some of them are briefly explained and discussed in further work.

Chapter 2

Background

This chapter provides a background of the key topics in the thesis. First I will give a brief introduction of the importance of robotics today and how robots are used in different warehouses. Then a background on the target storage system will follow. In the last section, relevant combinatorial optimization problems and how to handle intractability will be discussed.

2.1 Importance of robotics

2.1.1 Robots for various tasks

Today, robots play an important role in our everyday lives. With various degree of intelligence, they are present inside devices like our smartphones, washing machines, TVs and cars. Even though their capabilities have improved drastically in short time, we have all quickly become so accustomed with them that we mostly enjoy their presence and barely notice how they rapid they develop.

While the vast majority of us are benefiting from of the robotic technologies while using our devices, the industrial robots are working 24/7 assembling these exact same devices that we use every day. According to the International Federation of Robotics (IFR), 225,000 industrial robots were sold in 2014 [2]. That is a 27% increase from 2013.

As the industrial robots work on finishing the assembling of products, the goods are taken out to warehouses around the world, ready to be ordered and shipped away. Many of these warehouses use some sort of robotic technology to pick up and ship goods. In 2013 1,900 logistic systems was installed worldwide [3], 37% percent more than in 2012.

2.1.2 Warehouse systems

As the robots replace human manpower in warehouses, we see different systems are used being used all over the world. While some systems make their robots travel between racks to pick up requested goods, other storages have lower degree of freedom and their robots do not move as freely. With the exception of some physical limitations, only creativity sets boundaries for a potential storage design and different warehouses choose different systems for their needs and constraints.

There are many possible ways to set up a working autonomous warehouse system, and having robots move among racks of goods is one of them. However, leaving room for robots between racks leads to a lot of open space that could be used to store more goods. On the other hand, when the racks are placed closer to each other, items tend to get less accessible because other items must be moved before a specific pickup.

A typical trade off when storing huge amounts of data on a computer is *memory usage* versus *access time* to the stored elements. For instance storing elements in multiple multidimensional data structures will use more memory than when keeping the elements in a single list. It is however faster to access an element or a (sorted) subset of the elements if the data structure is well chosen, than iterating over the entire list for each different query. The analogy between a computer storage and a physical warehouse storage is clear; a storage with space between shelves will have an advantage over the dense system when it comes to pickup time. On the other hand, a company using a dense system will need a smaller warehouse and pay a lower rent.

2.1.3 What about our jobs?

A question that often arises when autonomous systems can do our jobs better than us, is whether the robots will take all our jobs and contribute to higher unemployment rates in the future or if they will help us do our jobs better. Although it is far from relevant in a technical and algorithmic point of view, it is important to discuss whether or not the results of this thesis can benefit the society.

Kevin Kelly wrote an article[4] where he argues that even though robot behaviors become more and more complex, it is not "a race against the machines" it is a "race with the machines". Ever since the industrial revolution, we have let machines take over jobs that they do more efficient than us. Looking at unemployment rates over the years, there is no trend indicating that any more of us are unemployed today than 30, 50 or 100 years ago.

A common argument for letting the robots take over the jobs they do better than us is that they do not replace us, but help us work more efficient. I am not going to debate this argument, but it is based on the experience we have today

and because we do not understand the robots future potential it is harder to argue otherwise. On the other hand, if we understood the full potential of the future robots, we would be able to predict which jobs robots would do better than us and see a potential for new human jobs. We might not be able to understand the robots full potential, but our history is full of examples where machines inherit our old jobs as we move seamlessly into new ones.

2.2 A compact box storage

2.2.1 Introduction

In the introduction, it was mentioned that the goal for this thesis is to investigate reorganization time in a storage with high density. Thereby, this storage falls into the category of memory winners and access time losers. The target storage system, is a particularly good example of a very high density storage. The storage is a grid consisting of stacks with objects. The objects are stacked upon each other such that only the top objects can be picked up and moved to other stacks.

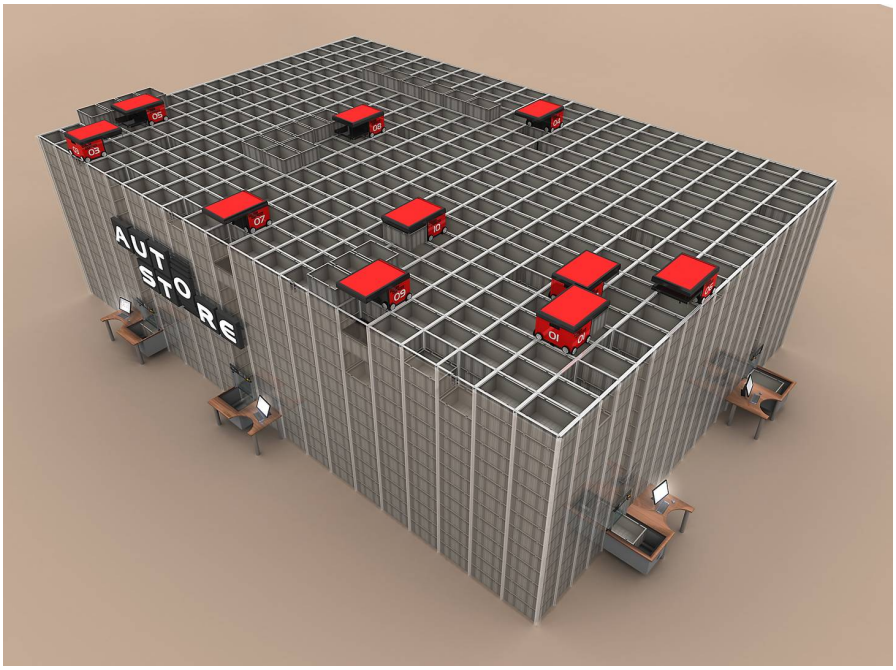

Figure 2.1: A grid with 20 × 20 stacks, 10 robots and 4 ports

Each stack is positioned firmly against its neighbors, thus no warehouse employee can operate between the stacks or have access inside the grid in any way.

Objects are only accessible from the grid's surface, where robots are working to organize the grid by moving objects between stacks. The robots are working autonomous and if an object is requested by a warehouse employee, a robot will have it delivered in one of the ports where an employee can pick it up.

2.2.2 Storage design

A typical storage design can be seen in figure 2.1. The storage has 20 × 20 stacks, each with a maximum capacity of 13 objects. The four ports on the grid's sides can be used by warehouse employees to pick up objects delivered by robots.

An important purpose of a dense storage system is to utilize space to the maximum. Not all warehouses have a rectangular or quadratic area available to place the grid on, so the grid must be customizable for almost all kinds of warehouses. To compress the storage to the full potential of a warehouse, the grid storage can basically have any possible shape. Figure 2.2 shows some possible grid shapes. Obviously, some grid designs are less convenient than others. The bridge in the bottom grid could easily become a bottleneck if robots have to travel over it all the time.

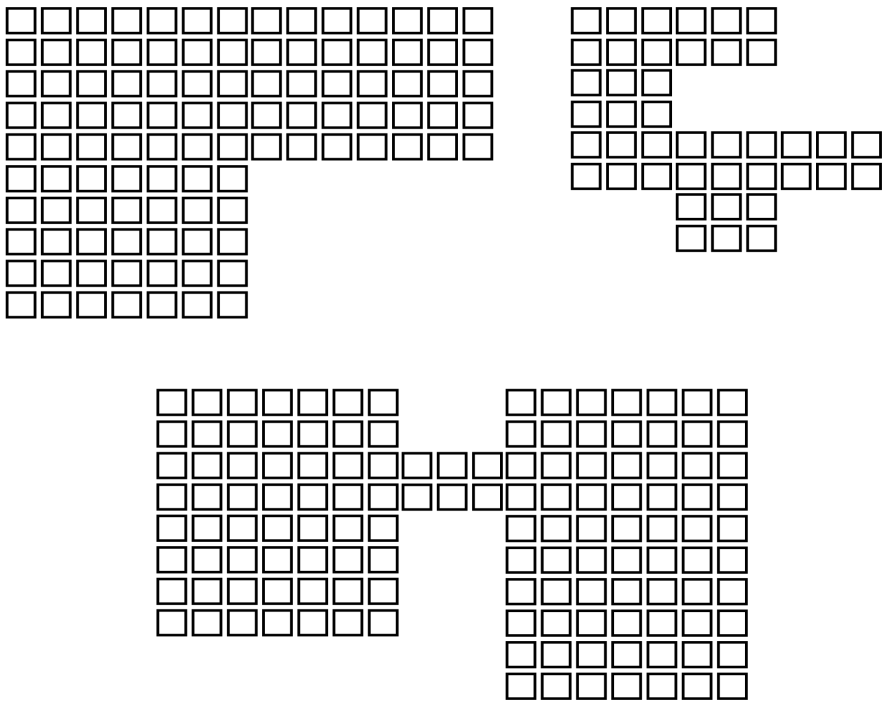

Figure 2.2: Three alternative grid shapes seen from above

A storage can also be built over obstacles or multiple floors in the warehouse

building. In the same way as the ocean water fills the sea bottom, a storage grid can fill the warehouse. By utilizing vertical space a grid can have different depths in certain areas. The varying depth is not a problem as long as the robots can move safely on a level surface. Some examples of grids with varying depth can be seen in figure 2.3. In the figure, two grids are built around physical obstacles, while the third one has an open space through it. The grids are viewed from the side.

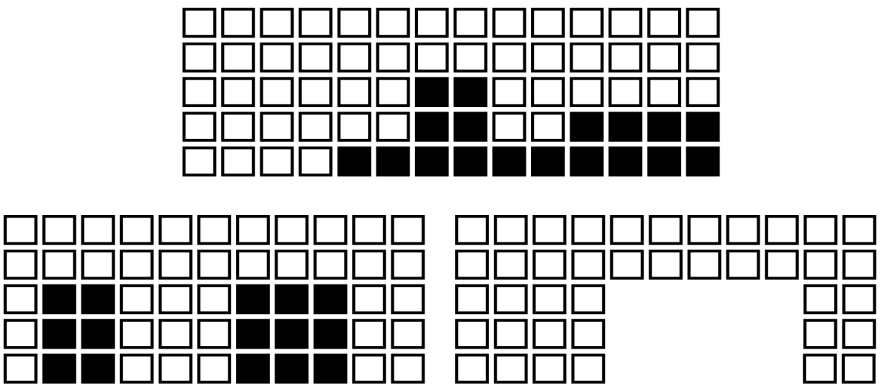

Figure 2.3: Grids built around physical obstacles or open space. Seen from the side.

2.2.3 Grid dimensions and fill rates

As different stores and factories have varying size of inventory, their storages also vary in size. The number of stacks can vary from small 10×10 grids to the largest 100×100 grids with a total of 10,000 stacks. The stacks capacity are usually somewhere between 10 and 30 objects, but in uneven grids like those shown in 2.3 shallow areas can have a capacity of only one object in each stack. The grids that cover large areas are usually also deep, hence the largest grids can contain as much as 300,000 objects.

Because utilizing available space in the warehouses is very important, the fill rate of each stack is high. As far as possible, there is only room for one or two more object on top of each stack on average. Due to this limitation, robots must unload objects all over the grid when they are digging for a bottom object. Thus, the high fill rate imposes some serious difficulties in planning where to move which object and when to move it.

2.2.4 The robots and object moves

On the grids surface, multiple robots are working in parallel to move requested objects between stacks and out to warehouse employees. Looking at the grid from above, robots can only move horizontally or vertically. Because a robot cannot move diagonally, a move from position (0,0) to (1,2) will require one horizontal moves and two vertical moves. A total of three moves. In figure 2.4 the three possible moves from (0,0) to (1,2) are shown.

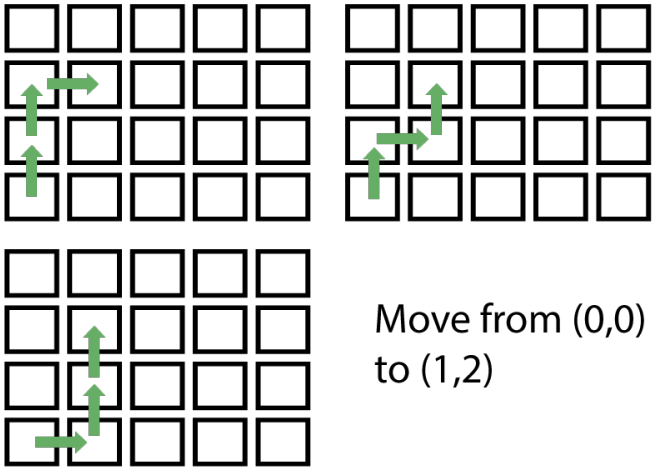

Figure 2.4: A move from (0,0) to (1,2). The grid is viewed from above.

The sum of total horizontal and vertical moves is known as *Taxicab geometry* or the *Manhattan distance*. Comparing the Manhattan distance of two or more different moves is believed to be a good indication of which move is the best. However, to get an even better comparison, robot speed, acceleration and move path must be considered. For instance, a move with many turns is more costly than a straight move. In figure 2.5 a move with three turns is illustrated together with a move with zero turns. The Manhattan distance of the left move is 3, while it is 4 for the straight move to the right. However, the right move is likely to go much faster because the robot only needs to start and stop once.

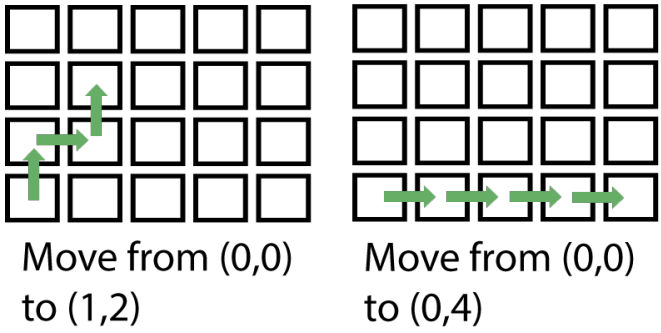

Figure 2.5: A move from (0,0) to (0,4). The grid is viewed from above.

In addition to speed, acceleration and move straightness, the pickup and drop off time of an object can vary dependent on stack depth. If the uppermost object in a pickup stack is located deep relative to the grid's surface, it takes longer time to get the object up. The same goes for deep drop-off stacks. Figure 2.6 shows two moves, where the shallow, right move is likely to be faster than the left, deep move.

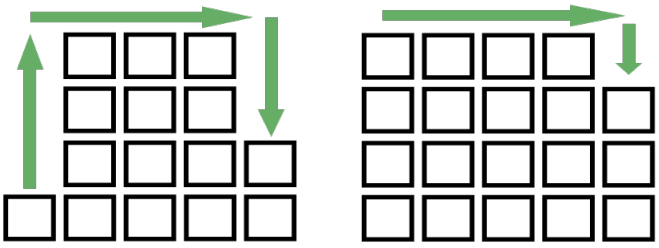

Figure 2.6: Left: Deep move. Right: shallow move. The grid is viewed from the side.

Because multiple robots are operating in parallel, many objects are likely to be requested at the same time and robots can block each other's way. What is pointed out in this section may be very relevant in theory, but not have the same significance in practice. Hence, it is hard to tell if every straight move always will be faster than one with many turns, or if a move with shallow pickup and delivery is faster than a deep one when we look at the big picture of a total reorganization.

2.2.5 Objects and storage behavior

In the target system, it is irrelevant what the different objects contain. An object could be a bin containing different goods or it could be one object. The normal behavior of the storage is to get various objects out to warehouse employees. As

objects are retrieved from the storage, it will get unordered over time, and the reorganization is a task that typically takes place during night time or at some other time when the storage is not very busy.

2.3 Combinatorial optimization

Combinatorial optimization is the discipline of finding an optimal object or set in a set of discrete objects or sets. The different problems of combinatorial optimization can be highly contrastive and includes for instance sorting, scheduling, planning and search. Examples of problems involving combinatorial optimization are *Minimum Spanning Tree*, the *Traveling Salesman Problem* (TSP) and the *Knapsack Problem*. The last two are known to be NP-complete (NPC) for the decision problem and NP-Hard (NPH) for the optimization problem. Combinatorial optimization problems often reside within the set of NPC and NPH problems, and exhaustive searches are not feasible on problem instances above a certain size.

2.3.1 Coping with NP-Completeness

It is the worst case complexity of an exhaustive search that makes these problems so hard to deal with. But life in general is not worst case, so usually there is no point worrying about worst cases and optimal solutions. Often, it is fairly simple to find a good enough solution.

A postman visiting multiple addresses located in different areas of a town is more likely to go for a route he is familiar with, than trying to find the optimal route every single day. Partly because he has found a route he likes, and partly because the fastest route is not *that much faster*. Of course, most postal services and transporting companies have some technical assistance such as GPS and route planning systems to calculate routes. However, because combinatorial problems are super complex, these planning devices cannot take every possible event into account. Therefore, some techniques can be used to provide really good suboptimal solutions in short time.

Random search

If we have a problem concerned with finding an optimal permutation, like the example above, just ordering the elements randomly can represent a search. By repeating this simple random search multiple times, some solutions will be better than others and it is possible to keep the best one. This method is totally based on randomness and running the search hundred, thousand or million times does not guarantee finding the optimal solution. However, randomness can be combined

with other, more qualified searches and help them explore an enormous search space.

Greedy search

A more qualified search is the greedy algorithm[5]. The greedy algorithm will always choose the local optimum at its current position before it moves on to the next node in the search. Such searches will not explore the whole search space, but they normally terminate faster than exhaustive searches. Performed on an n-clique[6] like TSP, to find an optimal permutation, the greedy search will terminate in quadratic time, $O(n^2)$. Although a greedy search in deterministic and basically always chooses the same path, variations can make multiple searches on the same graph yield different results.

The simplest way to explore search space further by using the greedy algorithm without any modifications is to change starting points. When searching an n-clique, run the greedy search n times, one for each node. By doing so, the search space is viewed a little bit different in each iteration which might lead to different solutions. Of course if the search require a certain starting point this is not possible, and some graphs might have some other properties that makes these searches pointless.

The ϵ-greedy algorithm is another, far more exploring, way to use random elements in a greedy search. With a small probability (usually 1 - ϵ) the algorithm chooses another path than the local optimum, and thereby varies the search a tiny bit. An ϵ-greedy algorithm can be implemented in different ways dependent on application. Repeating ϵ-greedy searches with different starting nodes can be a good way to increase exploration of the search space.

2.3.2 What about solving in parallel?

If these problems are so incredibly hard to solve, why don't we just solve them in parallel with multiple processors or multiple machines? In practice a parallel speed up might have a positive effect on the runtime on smaller instances of an intractable problem. However, theory shows us that parallelization cannot defeat the complexity of these problems [7].

Consider combinatorial problem where some optimal solution is a permutation of a set with eight nodes, ABCDEFGH. The total number of unique permutations are $8! = 40320$. Because this is a small instance of a certain problem, the search space is not enormous, but it works as an example. Assume we have eight machines, $m1, m2, ..., m8$, and that we store one letter to the first position of the permutation in each machine. Then $m1$ would generate permutations starting with A, $m2$ starting with B, and so on. Each of these machines now only have to search

through $7! = 5040$ permutations and return their individual result for comparison by a last machine. Great, from one machine searching 40320 permutations, we now have eight machines doing $\frac{1}{8}th$ of the work.

Unfortunately, using this technique each machine still solves an exponential subproblem and as the problem grows, $(n-1)!$ will not be an enjoyable number anymore. If the problem is in NPC/NPH, the subproblem that each machine solved is still in NPC/NPH. If we want to solve the problem in polynomial time, we could do so by letting $n!$ machines return one solution each. Then these $n!$ results can be compared, and the best solution will be found. Comparing $n!$ can be done in parallel logarithmic time, thus we have a polynomial solution to the problem.

But even though there is a theoretical way to solve such problems in polynomial time, we would need $n!$ machines to do it. According to Universe Today[8] there are approximately 10^{80} atoms in the universe. If we wanted to solve a problem instance with 59 nodes using the suggested polynomial approach, we would first have to make $59! \approx 1.4 \times 10^{80}$ machines. One atom for each machine.

Chapter 3

Methodologies and experiments

There are numerous ways to solve the combinatorial reorganization problem. A classical exhaustive or brute force search would find the optimal solution and provide a proof that the best solution is found. However, when working with large grids, performing an exhaustive search is very likely to require an enormous amount of memory (see section 3.2.1) or take too long time to complete. In contrast to a deterministic exhaustive search, algorithms based on randomness can very well be used to deal with complex, combinatorial optimization problems. The methodologies considered here will reside somewhere in between these two extremes.

This chapter covers different methodologies, implementations and experiments. Section 3.1 describes the problem definition as well as properties and constraints related to the problem. Then, the following sections discuss different methodologies, implementations and experiments to solve the reorganization problem. Some hypotheses and experiments may seem unnecessary because their performance is almost obvious. However, those experiments will lay a foundation for creativity and new ideas. Of course, some possibly relevant methodologies will be left out for the benefit of being able to focus on some particular ideas.

3.1 Problem definition and introduction

The reorganization problem in this thesis will consider the grid as a *static* environment. This means that nothing else is interacting with the grid during the reorganization. A problem instance of the reorganization problem consists of an initial storage and two lists, L1 and L2. The initial storage is a grid in a given state, and the goal is to reorganize the grid to meet the requirements of the two lists.

The highest prioritized list, L1, consists of objects that should be on the grid's

surface after reorganizing the storage. This implies that the size of L1 needs to be less than or equal to the surface of the grid, $|L1| < X \times Y$.

The second list, L2, consists of objects that have lower priority than all objects in L1, but higher priority than the rest of the objects in the grid. No objects in L2 can be positioned higher in the grid than an object in L1. No objects in L2 can be positioned lower than any non-prioritized object (i.e. an object not in L1 or L2).

Figure 3.1 illustrates an example of an initial grid. The red objects belong to the highest prioritized list, L1, while the green objects belong to L2.

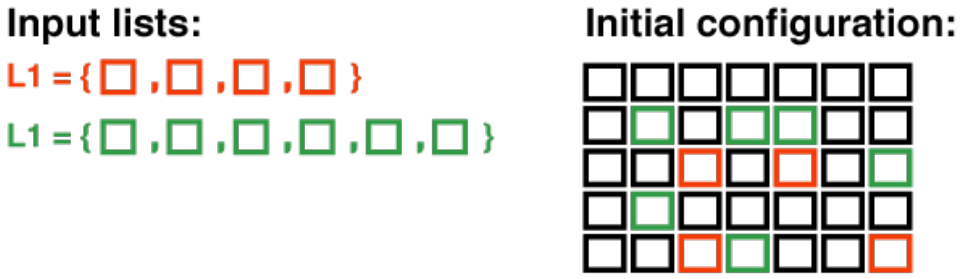

Figure 3.1: Initial grid, seen from the side as seven separate stacks

The reorganization is finished when no objects violate the rules described above. As long as the requirements of L1 and L2 are fulfilled, it does not matter which stacks the objects are located in. This property implies that there are many possible final configurations, sharing the property that they have the same number of each object type on every level. Figure 3.2 illustrates some possible final configurations.

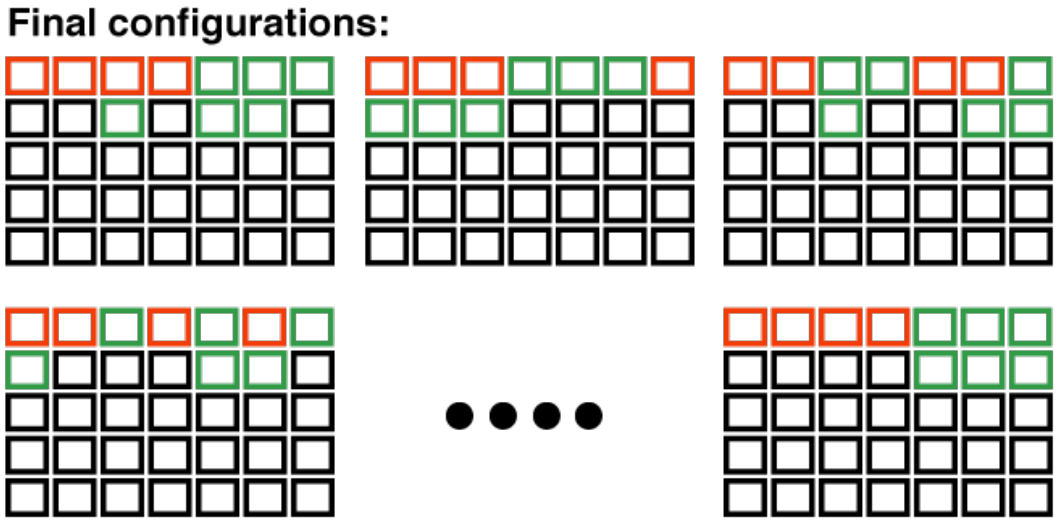

Figure 3.2: Final grids as seven separate stacks. Viewed from the side.

3.1.1 Solvability

Based on the given problem definition, the grid's solvability depends on two things:

- All bottom objects must be reachable. I.e. the empty space in the grid, $|E|$, must be greater than or equal to the stack with the highest capacity.
- The size of $L1$ must be smaller than or equal to the area of the grid, $X \times Y$.

As long $L1$ does not cover more than the surface of the grid and it is possible to dig up all objects, the grid is solvable. In practice, either $L2$ or $L1$ could be empty. If both lists were empty, we would have an already solved instance of the problem.

3.1.2 Initial and final grid

The previous section showed that there are many different solutions to one initial grid, and that all these possible solutions share the property that they have the same number of objects of each type on each level. This property makes it possible to keep track of how close we are to a solution. Figure 3.3 illustrates how differently prioritized objects are positioned in the solved grid.

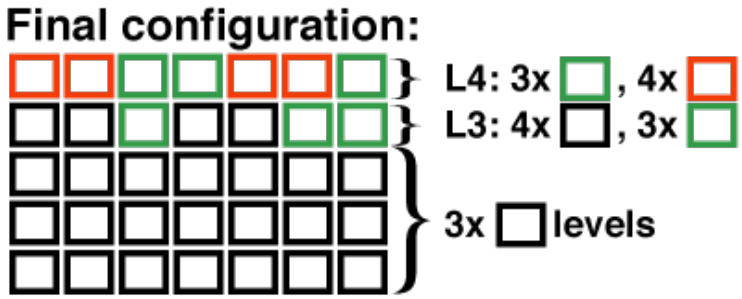

Figure 3.3: Common properties for all solutions. Viewed from the side.

Again, it is not important which stack each object is placed in, but rather on what level the different objects should be. During reorganization, each move should (as far as possible) get the grid closer to a solution. In figure 3.4, the numbers on the right hand side of the grid illustrates how many objects are correctly positioned and how many objects a solution should have on each level.

Figure 3.4: Initial configuration and object positions on each level

With this information, it is possible to write a module that can support the algorithms when choosing what object to move where. Every time an object is popped from one stack and pushed onto another, the module is updated with correct information, just like the figure above. Section 3.4, describes how this support module works and why it is important.

3.1.3 Algorithm output and measures of optimality

Given a problem instance, an algorithm will find a set of moves that takes the grid from an initial configuration to a final configuration. One must not confuse the

algorithm with the system that organizes the physical storage or a simulation of the physical system. The algorithm's only task is to find a set of moves that the storage and its robots can carry out.

Given an initial grid (G) and two lists (L1 L2), algorithm (A) will find a set of moves (M) that transforms G from its initial configuration to a final configuration.

When the algorithm terminates, it will either decide that the grid is unsolvable or return a set of moves that transforms the grid from the initial configuration to the final configuration. Challenges related to fill rates, moves and robot concurrency were presented in sections 2.2.3 and 2.2.4. How these factors affect the total reorganization time of the storage is a complex matter, and measuring the optimality of the algorithm's output is not trivial. By sending algorithm outputs to a real or simulated system and reorganize the storage, the total reorganization time of each run will form a basis of comparison to decide which factors are most important.

A high total Manhattan distance and few moves may reorganize the storage faster than low Manhattan distance and many moves. Every single move has some overhead (such as getting to pick up stack, pickup/delivery depth, navigate between other robots, etc.), that argue in favor of reducing the number of total moves instead of the total move distance.

These parameters can be tuned easily, and weights or costs can be added to implement realistic methods to calculate optimality. Therefore, this thesis will have a theoretical focus and only use Manhattan distance and number of moves as measures of optimality. It is still possible to create an optimal function for optimality later.

3.1.4 Keeping the robots busy

As robots are cooperating on the grid's surface, it is important that they are kept as busy as possible while they do not get in the way of each other's work. A set of moves can easily be returned as a sequential list where each move must be finished before the next one can be done. This would work well for storages with only one robot, but for multi-robot storages this is not very efficient.

The initial sequential list of moves must be converted to a list with dependencies such that multiple moves can be done in parallel. Along with factors like total number of moves and total distance, the parallelizability of the sequential output list is also utterly important so the robots are busy as long as the final configuration is not reached. Section 3.7 discuss this topic.

3.1.5 Scalability

Although this problem is concerned with reorganizing the grid based on the two lists, it should be straightforward to use the algorithms in this chapter on a problem with more than two lists. In fact, if each object was assigned its own unique priority such that there was one list for each object, the problem can still be solved using the algorithms suggested here. However, the complexity would increase drastically. In section 3.2.1 the combinations of a given grid is discussed.

3.2 Complexity analysis and exhaustive search

The background chapter explained how life in general is not worst case, and that we rarely need to find the optimal solution. However, if an exhaustive search is feasible and terminates within acceptable time, obviously that would be the best algorithm to solve the problem. In this section combinatorial complexity issues and exhaustive search are discussed.

3.2.1 Computational Complexity - Permutations and combinations

As an introduction to complexity analysis, looking at combinations [9] of the grid seems like an appropriate place to start. Assume we have a grid consisting of four stacks with three objects in each. The objects are represented by their IDs; ABC..., and so on. An example grid is illustrated in figure 3.5. Both as four separate stacks and in string representation.

Figure 3.5: Four stacks with IDs (A-L)

ABC;DEF;GHI;JKL

Figure 3.6: Four stacks represented by IDs as a string

If we ignore the semicolons in figure 3.6, the grid has $12! \approx 479.000$ permutations. But the semicolons are relevant because they represent a second complexity dimension. The first dimension is the order ABC..., and the second dimension is where the stacks are separated by a semicolon. For instance DABC;EF;GHI;JKL is a different permutation than the one above because D is on top of the first stack, not the second. Dimension one, represented as a string of IDs, have 12! permutations as explained above. The second dimension takes object-stack affiliation into account and increases the number of permutations.

When looking into dimension two, let's first assume for the sake of simplicity that the stacks have unlimited capacity. This means ABCDEFGHIJKL;;; is a valid permutation with all objects in stack one. In this case, the number of possible permutations can be expressed as $12! \times \binom{15}{3} \approx 2.18 \times 10^{11}$. To explain this, assume we have one of the 12! permutations in dimension one, then we simply choose three spots to place the semicolons ($\binom{15}{3}$ can be read aloud as "15 choose 3"). These three spots can be chosen in $\binom{15}{3} \approx 455$ different ways. Hence, for each of the 12! permutations, we have 455 ways to put in the semicolons.

But the stacks will have a maximum capacity, therefore all variants of "15 choose 3" will not be valid. For instance, appending all semicolons at the end of the string will not be a valid variant unless the leftmost stack has capacity to contain all the grids objects. When working with a limit on stack capacity, only a subset of $\binom{15}{3}$ are valid configurations. This subset can be found by looking at the different distributions of objects in stacks. In the current example with 4 stacks (S1 to S4) with maximum capacity 4 and a total of 12 objects, we have the following distributions:

S1	S2	S3	S4	-	P
4	4	4	0	$\frac{4!}{3!}$	4
4	4	3	1	$\frac{4!}{2!}$	12
4	4	2	2	$\frac{4!}{2!\times 2!}$	6
4	3	3	2	$\frac{4!}{2!}$	12
3	3	3	3	$\frac{4!}{4!}$	1
-	-	-	-	SUM	**35**

Table 3.1: Valid object distributions in a grid with finite capacity

Hence, the total number of unique permutations (P) are: $12! \times 35 \approx 1.68 \times 10^{10}$.

So far, IDs has been used as permutation elements. In fact, we do not care about the object IDs, except when moving objects to keep track of where all the objects have been moved at all times. IDs are not relevant because, in the organi-

zation problems perspective, it does not matter if two objects of equal priority is swapped in the permutation. By using object priorities as permutation elements instead of object IDs, clearly we will achieve a great reduction in number of unique permutations. Assume the different objects are distributed like in the grid below. Remember, red represents $L1$, green $L2$ and black objects have the lowest priority.

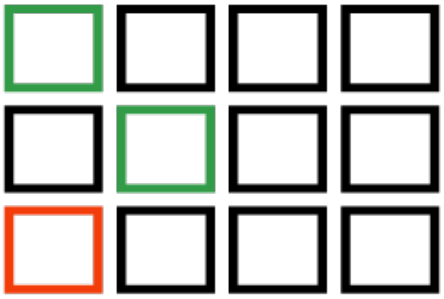

Figure 3.7: Four stacks. Objects represented by their value color.

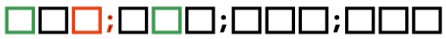

Figure 3.8: Four stacks in string representation. Object priority as value.

Looking at the first dimension, i.e. without semicolons, we have $\frac{12!}{9!\times 2!\times 1!} = 660$. Adding the second dimension, capacity and object-stack affiliation, with the same capacity as earlier, the total number of permutations are: $\frac{12!}{9!\times 2!\times 1!} \times 35 = 660 \times 35 = 23100$.

This is definitely a great reduction, but a realistic grid has many more bins. It is more likely to be 100k objects in a simple grid which means 100k!+ ID-permutations. Although the number of value/priority permutations will be much lower, the number of permutations is a completely unmanageable size. By looking at the priority-permutation instead we'll get a smaller number, but still not a number we would like to deal with in terms of combinatorial optimization.

Below are the numbers for a 5x5x5 grid, i.e. 25 stacks and a total of 125 objects, and assuming stacks with infinite capacity. Keep in mind that this grid is also highly undersized. *24* represents the separation of stacks (the semicolons).

ID:
$\frac{(125+24)!}{24!} \approx 6.14 * 10^{236}$ unique permutations.

Priority/value (distributed 50/50/25):
$\frac{(125+24)!}{(50!\times !50!\times 25!\times 24!)} \approx 4.28 * 10^{82}$ unique permutations.

3.2.2 Computational Complexity - Exhaustive Search

In the previous section, examples showed that the number of unique permutations in a grid gets unpleasantly large. Even when working with a tiny grid or just considering permutations with respect to priority, the numbers get enormous. These results strongly convinces us that this problem is intractable with a brute force search. However, it feels appropriate to show an example by designing a brute force algorithm and view some results.

Exactly like a classical puzzle problem, identical *states* are reachable from multiple paths. By looking at the state, it is not possible to determine what moves have been made (if any). Due to this property, visited stated must be stored. When reaching an already stored state, the state with the shortest path is stored. To make sure the exhaustive search terminates, the search is stopped when a solution is found. This prevents the search from finding neighboring solutions, but such a solution would be inferior anyways (because it would require more moves).

Allowing a few moves in any grid will result in many new states. In order to keep track of how the number of states grows and to prevent the search to continue into long unnecessary paths, the brute force algorithm uses an iterative approach. By first allowing one move, then two and so on, it ensures that the search will never perform more moves than the number moves required to find a solution.

3.2.3 Pseudo Code for Exhaustive Search

The iterative approach is divided into two parts. The first part starts the brute force algorithm with the number of allowed moves as a parameter. As long as no solutions are found (i.e. the *solution* set is empty), the number of allowed moves increases by one and the brute force algorithm is started over.

The brute force algorithm creates all possible permutations with the allowed number of moves. If a solution is found, the solution is stored and (as mentioned) when the solution set is not empty any more, then algorithm terminates.

This iterative approach prevents the algorithm from searching for solutions that require more moves than the optimal solution. This means that not all states will be visited, but only the necessary states to find a solution. It is still a complete brute force search with a restriction on number of moves.

Algorithm 1 Iteratively start a search with max moves

```
1: function IT-BRUTE-FORCE(maxMoves, states, solutions)
2:     while solution.isEmpty() do
3:         maxMoves+ = 1
4:         grid.BRUTE-FORCE(maxMoves, states, solutions)
5:     end while
6:     Return solution
7: end function
```

Algorithm 2 Performs all possible moves in the current state

```
1: function BRUTE-FORCE(maxMoves, states, solutions)
2:     if solved then
3:         solutions.add(this)
4:         Return solutions
5:     end if
6:     if moves ≥ maxMoves then
7:         Return
8:     end if
9:     copy = grid state with stacks
10:    for stack1 in copy.stacks do
11:        for stack2 in copy.stacks do
12:            if stack1 ≠ stack2 then
13:                stack1.push(stack2.pop())
14:                if new or better state then
15:                    states.add(this)
16:                    copy.BRUTE-FORCE(maxMoves, states, solutions)
17:                end if
18:            end if
19:        end for
20:    end for
21: end function
```

3.2.4 How do the number of states grow?

One simple move in an initial grid will generate $n \times (n-1)$ possible new combinations. In the example with a grid of four stacks we have $n = 4$, so we would get $4 * 3 = 12$ new states from one initial move. The next move from these 12 states will generate $12 \times (n-1)(n-1) = 12 \times 3 \times 3 = 108$ new states. This is an exponential growth, and for the first two steps the search space grows with $(n \times (n-1))^m$ and $((n-1) \times (n-1))^m$, where m is the number of moves. However, following one of these functions, already visited states will be found again. Thus, the state space cannot grow at such a high rate.

Figure 3.9: One simple initial grid generates 12 new states

Figure 3.9 shows the first 1 + 12 states of an exhaustive search on a tiny grid. Results from exhaustive search runs on this one and larger grids can be found in section 4.1.

3.3 Random algorithms

As we have could see in the previous section and exhaustive search is probably not the most clever way to solve this problem. Instead, we will look briefly into random algorithms and how randomness can be used in combination with more deterministic algorithms.

3.3.1 Totally random movements

Random algorithms that pop and push objects until a solution is found was tried out with very small grids, but because of the problems nature, we will not spend much time with this. Instead, let us look at an experiment related to section 3.2.4. Assume we start off with the same initial grid. Then one of the 12 random moves is made randomly. After that, another random move is made. And so we continue until a solution is found. For each random move, there are $n \times (n-1)$ possible new moves, given we do not remember any states and use that history.

Assume a solution could be found after 20 moves. Then we would have to perform 20 random moves, each with $n \times (n-1)$ choices. It is clear that our odds of finding that solution are not very large. Although, there are multiple final

states, we are more likely to end up somewhere else than in a solution after 20 random moves.

3.3.2 Partially random algorithms

Even though 100% random algorithms will not work for this problem, the combination of deterministic and random algorithms could be powerful. Randomness is a very good tool if we want to explore the search space for instance. Because an exhaustive search will be infeasible, we will have to make sure we explore the search space and not just go greedy down one branch of the search tree. For instance, when looking for a place to put an object, choosing the nearest stack is not necessarily the best option for the future. In the following sections, algorithms with some randomness will be introduced.

3.4 Grid configuration

Exhaustive or totally random searches do not seem feasible and the object moves must be done more wisely. Section 3.1.2 mentioned that a supporting module can help the reorganizing algorithm to know where differently prioritized stacks should be. In this section I will introduce a supporting data structure called *GridConfiguration* that keeps track of all objects positions and helps the working algorithm make qualified object moves.

3.4.1 Object priorities and levels

Because there are three different object priorities and a finite space these objects can fill, calculating what kind of object should be located on which level is trivial. Using the ocean water analogy from 2.2.2, we can simply "fill" the grid with non-prioritized objects, then L2 objects and finally L1 objects, and we have a valid final configuration. Counting the number objects from each priority on every level gives us the final configuration level by level.

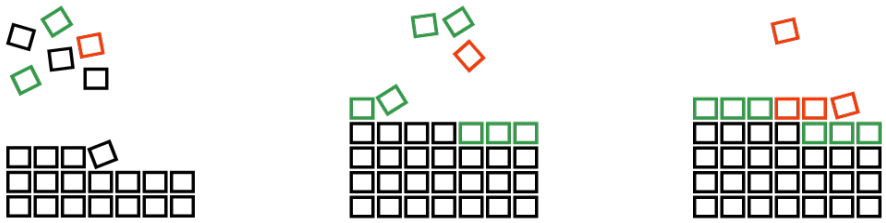

Figure 3.10: Filling the grid by priority

3.4.2 Stacks, mismatches and updating

Now that we know the valid final configuration on each level, we also know whether an object should be removed from a stack or not. An object positioned on a level where it should not be located, is called a *mismatch*. When working with a certain stack, we continue to pop objects from it as long as any mismatch is found in the stack. In figure 3.11 the stacks in the left grid are marked where the lowest mismatch is, based on the final grid to the right. Objects below the lowest mismatch point will never need to be moved.

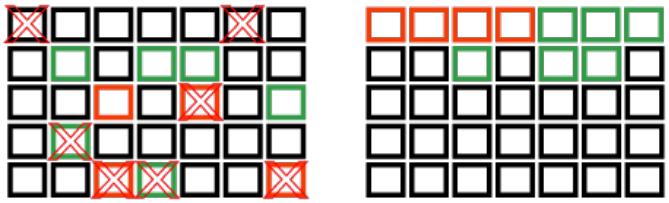

Figure 3.11: Lowest mismatches are marked in the left grid.

As objects are being popped from one stack and pushed onto another, the GridConfiguration changes accordingly. By keeping the GridConfiguration up to date at all times, redundant moves can be avoided and the storage gets reorganized much faster. In the next section a base algorithm using the GridConfiguration module is presented. Figure 3.12 shows how the mismatches are updated from figure 3.11, after some moves are made. Notice that two stacks (2 and 5) are finished and that mismatches can occur in top of finished substacks (1 and 3).

Figure 3.12: The left grid is now 'matching' the final solution better.

3.5 DigFill - The base algorithm

When dealing with stacks as a data structure, there are simply no shortcuts in terms of how to get a covered object up to the top. Each unsolved stack has to be dug up until it is valid. Then the stack will be filled up with valid objects. Having

that said, it is possible to do a lot of detours. Every moved object could potentially have to be moved at a later point in order to clear space for other objects or to fill another stack. While length of moves will be dealt with in section 3.6, this section will provide a base algorithm as a framework to reorganize the grid.

Regardless of what methodologies or techniques that are used from now on, the base algorithm relying on GridConfiguration will execute all moves between stacks. This will ensure that every move is consistent with GridConfiguration and that the grid eventually ends up in a final configuration.

The base algorithm, *DigFill*, is composed of two steps that represent the core functionality of the algorithm. All stacks with a mismatch are put in a *digQueue*. The digQueue is kept up to date at all times by removing solved stacks and adding unsolved stacks. Until the grid is solved, the algorithm repeats the dig-fill cycle; pick a *digStack* from the digQueue and repeat the two following steps:

- **Dig:** Objects on the digStack are popped and unloaded on other stacks as long as the digStack contains invalid objects. Whether an object is invalid or not is decided by GridConfiguration.
- **Fill:** The digStack is then filled with valid objects. If no valid objects are found on top of any other stack, the filling stops.

Remember that every time an object is popped from one stack and onto another, the GridConfiguration (see section 3.4) is updated instantly.

3.5.1 Pseudo code for DigFill

Algorithm 3 DigFill

```
1: function DIG-FILL(initialGrid)
2:     digQueue.add(all unsolved stacks in initialGrid)
3:     while grid not solved do
4:         digStack = digQueue.getStack()
5:         digStack.dig()
6:         digStack.fill()
7:         digQueue.add(all unsolved stacks in grid)
8:     end while
9:     Return solution
10: end function
```

3.5.2 Digging

Every stack in the grid, depending on what kind of objects it contains and how the objects are ordered, will contain a valid substack. A substack starts from

the bottom, and stops where a mismatch is found. If the bottom object is a mismatch, then the valid substack is an empty stack. On the other hand, if there is no mismatch in the stack the entire stack is valid. Digable and non-diggable areas are shown in figure 3.13.

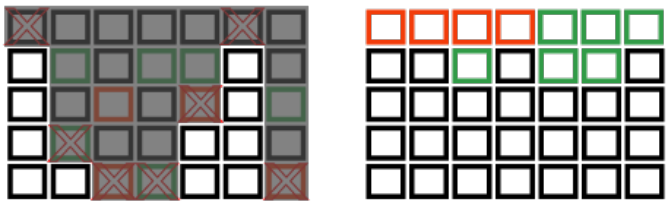

Figure 3.13: The non-shaded area shows valid substacks.

Before the dig-fill cycle starts the digQueue is updated with invalid stacks. The first thing to happen in the dig step is to choose a digStack randomly from the stack The dig step's task (line 5: Alg. 3) is to always dig until the entire stack is valid, i.e. until the valid substack is reached. Each object popped from the digStack needs an unload stack to be pushed onto, section 3.5.5 explains how these stacks are chosen.

3.5.3 Filling

The second step in the dig-fill cycle is filling up the digStack again (line 5: Alg. 3). When the filling starts the digStack is valid, and it should only be filled with other valid objects. If no valid objects can be found on the surface or the digStack is saturated (i.e. any object pushed onto it will make it invalid), the fill step stops.

During the fill step, a digStack might accept multiple priorities. For instance a digStack may accept non-prioritized objects, objects in L2 and objects in L1. In this case the base algorithm will first look for an unprioritized object, then an object in L2, and finally, if none of the two lower prioritized ones were found, an object from L1. So the fill order is; non-prioritized, L2-objects, L1-objects. Keep in mind that the distances between stacks are not considered yet, and algorithms with a focus on total Manhattan distance will ignore this order and look for the closest stack with a valid fill object.

Assume stack number 1 is being filled in figure 3.14. In this situation choosing either a black or a green object will preserve stack validity. The DigFill algorithm chooses the black one because it has the lowest priority.

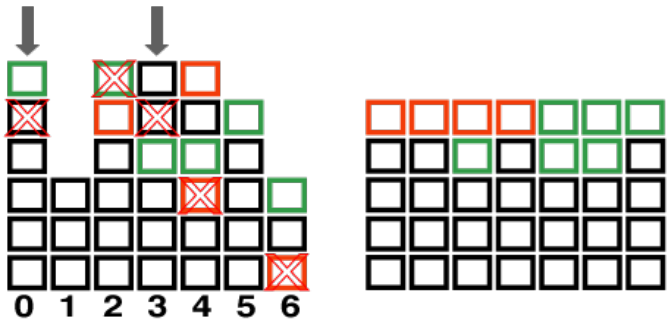

Figure 3.14: Which stack should be chosen as filler?

A potential upcoming challenge is now where to find fill objects and where to unload the excavated objects. This issue is clarified together with the unload issue in section 3.5.5.

3.5.4 Selecting digStack

As a dig-fill cycle ends, new invalid stacks are added last in the digQueue. When a new cycle starts, a new invalid stack is drawn randomly from the digQueue. Various attempts to sort the queue and dig stacks with a higher priority first has been made. Two of them will be discussed here.

The mismatch depth, i.e. the lowest invalid stack, can be used to compare the urgency of different stacks. The first attempt to pick digStacks in a certain order, is to sort the digQueue based on the stacks lowest mismatch point. The stack with most invalid objects will be dug first. A reverse sort is also possible and was tried out as well.

The queue was also sorted using *fill opportunity* based on the available objects on top to fill the stack after it was dug. A high fill opportunity is when enough objects of each priority is available on the surface to fill the stack after it has been dug up. A low fill opportunity is when few or no objects are available to fill the stack after a stack has been dug up.

None of these orderings showed any great improvements in terms of number of moves to reorganize the grid. Comparison of different orderings can be found in section 4.2.8.

3.5.5 Versions

So far, how unload stacks and fill stacks are chosen has not been mentioned, and by now objects are unloaded and picked from arbitrary stacks when digging and filling. The first implementation of DigFill used this approach.

However, popping another valid stack to fill the digStack is a redundant move. When a valid stack is popped, it needs to be filled with an object that maintains validity again. From version 2 this issue is improved. If no fill object can be found on any invalid stack, the filling stops.

Unloading objects on any random stack is also a bad idea if there exists another stack that needs the object already. This means that if there is a stack that can take the popped object and remain its validity (and thereby take the grid closer to a solution), unload on that stack. In version 3, this is improved by first looking for stacks that maintain validity when the object is unloaded on them. If no such stack is found, the object must be unloaded somewhere else and the DigFill algorithm just picks a stack with capacity. Section 4.2 shows results from simulations with different DigFill versions on grids of various sizes.

Version	Unload	Fill
V1	Random not full stack	Find valid objects on any stack
V2	Random not full stack	Same as V1, but cannot pick objects on solved stacks
V3	Look for a stack that needs the popped object	Same as V2

Table 3.2: Versions of DigFill

3.5.6 Iterative DigFill

Because digStacks are chosen somewhat randomly from the digQueue, running the algorithm twice on the same grid might not give the same result in terms of moves and total Manhattan distance. Sorting the digQueue was discussed in section 3.5.4, but results from simulations have shown that the ordering influences the result almost nothing (see section 4.2.8).

Section 2.3.1 discussed how to deal with randomness. A first approach is to run DigFill a number of consecutive times, to get a set of solutions to choose from instead of just one solution. In section 4.2 this is exactly what is done. This approach is well suited to report results because we generate a basis for statistics and it can be used in an actual system to choose the best among the solutions.

3.6 DigFill with greedy search

All attention has so far been directed towards the moves so far, and the total Manhattan distance have not been of much importance. Although reducing number of moves is important because there is a lot of overhead for each move, it is still important that moves are made as locally as possible. In this section a greedy search for the nearest unload and fill stacks will be applied to the DigFill algorithm in order to make local moves.

Back in section 3.5.3, a fill priority order in a stack was suggested. It was also mentioned that later on, we will ignore this order for the benefit of distance. This will be tried out now, so finding the closest stack is now more important than the order of object priorities. However, the DigFill cycle still decides that the fill objects must be valid.

The same goes for unloading objects when digging. Because the popped object only have one value (priority), there are not multiple priorities consider here. Still, the dug object should be pushed onto a nearby stack as far as possible.

3.6.1 A sorted list of neighbors

The implementation of greedy algorithm is simply a data structure for each stack that contains a sorted list of its neighbors. For a grid with 100 stacks, this will create $100 \times 100 = 10.000$ pointers - each stack has a list with 99 pointers to its neighbors. For a grid with 10.000 stacks, we will get $10.000 \times 10.000 = 100$ million pointers.

So the implementation is straight forward. Simply, for each stack, create a list of all other stacks and sort it by Manhattan distance to the current stack. When looking for a nearby stack to push an object to or pop an object from, simply iterate through the list (which is already sorted) and pick the first that meets the requirements of what object priority needed. An example of a grid and two lists can be seen in figure 3.15.

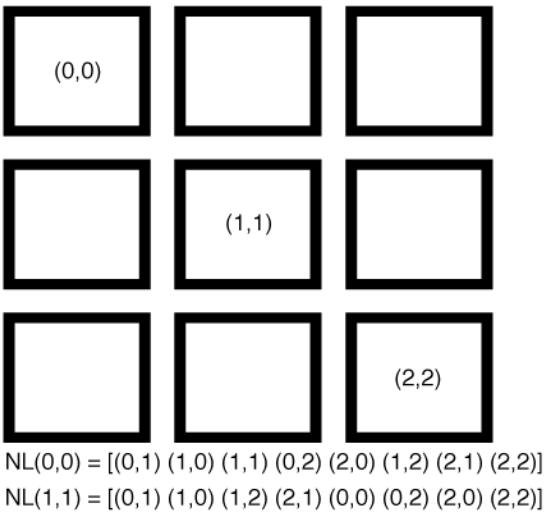

Figure 3.15: A simple grid with two examples of neighbor lists. Seen from above.

Although this method only needs to be initialized once and runs in linear time once initialized, this method requires huge amount of memory as the total number of stacks increase. Section 5.2.3 suggests a better way to implement it.

3.7 Parallel moves

Until now, results have been measured by number of moves or Manhattan distance when estimating the optimality of a solution, and the set of final moves has been represented as a sequential list. Because the robots are working in parallel to solve the problem, they should be working nonstop from the first pickup to the last delivery. To make sure the robots can do that, we need to ensure that the moves in the sequential output from the algorithm can be parallelized.

This section discusses dependencies between moves and how the robots can know when a move can be performed. To clarify; this section is not concerned with whether or not an algorithm can solve the current problem in parallel, it simply discusses how the sequential output can be prepared such that robots can perform moves concurrently.

Although parallelization is not the main scope of this thesis, and the algorithms implemented so far are designed to return a sequential list of moves, it is essential to show that these sequential lists can be parallelized.

3.7.1 Move dependencies

A move is an operation where a robot picks up an object from a pickup stack and moves it to a delivery stack. First, we will look at the move as one single operation, where pickup is immediately followed by delivery. Because a move uses two stacks, a move will have two dependencies - a dependency on the pickup stack and a dependency on the delivery stack. Below, figure 3.16 illustrates a reorganization of a 6-stack grid.

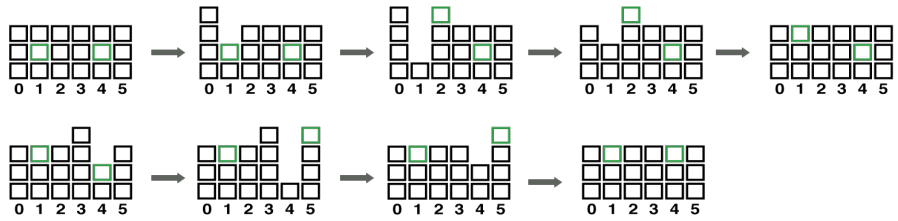

Figure 3.16: An eight move solving

A robot performing these moves in a sequence will get the grid from its initial state to a final state in 8 moves. In a small grid like this one, one robot can do the job alone, and multiple robots would probably be redundant in a grid of this size. Thus, assigning dependencies is unnecessary in a scenario with one robot. As we will see later, realistic grids are much larger with multiple robots, and when robots are working together they need to know when and where an object can be picked up and delivered.

As stated earlier, a move is dependent on two things; the pickup stack and the delivery stack. Hence, the last move that either picked up or delivered an object from/to one of the current move's stacks is a dependency to the current move. E.g. the first move (move from stack 1 to stack 0 in figure 3.16) has no dependencies because no objects has been picked up yet, the second move has the first move as one of its dependencies because it is picking up in stack 1, the third move has the first and the second move as its dependencies, and so on.

The table below shows a complete list of dependencies. The first number is the move number, the two next are pickup and delivery stack separated by a hyphen and the two last numbers are pickup and delivery dependency. No dependency is marked with a hyphen, -.

Move	From-To	Dep1	Dep2
0	1-0	-	-
1	1-2	0	-
2	0-1	0	1
3	2-1	1	2
4	4-3	-	-
5	4-5	4	-
6	3-4	4	5
7	5-4	5	6

Table 3.3: Table with dependencies

A directed acyclic graph (DAG) to represent moves with dependencies can easily be drawn. The ingoing edges vertices (moves) represent dependencies. This is a fairly simple example, and solutions on larger grids will have much more complex graphs. The graph in figure 3.17 illustrates how moves can be parallelized. Each subgraph has four moves and these subgraphs can be finished in parallel.

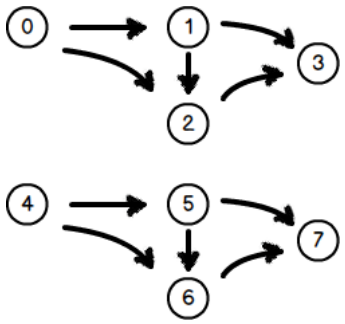

Figure 3.17: A directed acyclic graph with dependencies

3.7.2 Pickup and delivery dependencies

In the previous section, dependencies was set to moves such that the full move had to be carried out before the dependency could be removed. This is an unnecessary exaggeration because the dependency between two moves only depends on the *action* (a pickup or delivery) that is performed on the mutual stack. For instance, move number 1 only depends on a pickup in move number 0. As soon as a robot have made that pick up, move number 1 can start.

Although some moves can be started before all of their dependencies are cleared, these special cases will not be taken into account. As an example, move number 3 is ready for pickup after move number 1 is delivered. However, if a robot makes the pickup in move number 3 right after move number 1, it might have to wait

for move number 2 to finish. It is possible to calculate the remaining wait time in order to make move number 3 just in time to overlap perfectly with the delivery of move number 2, but due to very complex robot behavior and a risk of ending up in deadlocks, a move needs all of its dependencies to be resolved before the move can start.

Although a move still have to wait for both of its dependencies to be cleared, a move dependent on a pickup action can start much earlier than if it had to wait for the whole move to be completed. In table 3.4, pickup action dependencies are marked with a *P* and delivery action dependencies marked with a *D*.

Move	From-To	Dep1	Dep2
0	1-0	-	-
1	1-2	0P	-
2	0-1	0D	1P
3	2-1	1D	2D
4	4-3	-	-
5	4-5	4P	-
6	3-4	4D	5P
7	5-4	5D	6D

Table 3.4: Table with pickup and delivery dependencies

When dividing each node into two subnodes, one for the P-action and D-action, dependencies can be illustrated in an *action graph* like figure 3.18. The directed edges point to moves from actions within preceding moves. It is now possible for move number 1 to start as soon as pickup in move number 0 is done. Move number two can start when pickup in move number 1 is done. This allows for a smoother overlap between actions than the move dependencies from section 3.7.1.

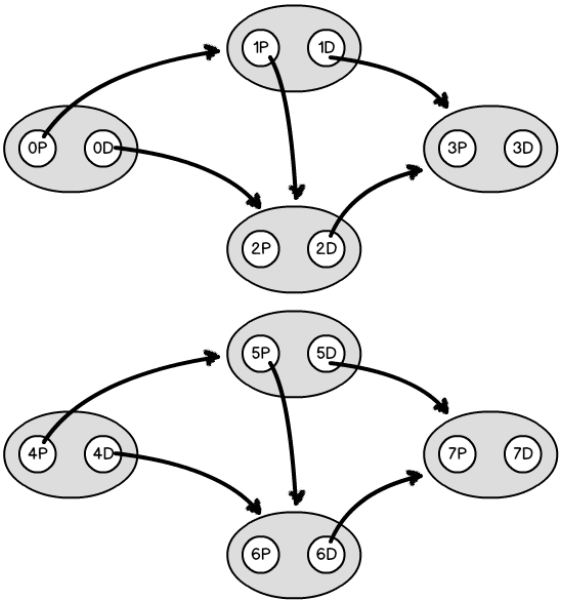

Figure 3.18: A directed acyclic graph with action dependencies

3.7.3 Parallelizability

Now that two different dependency methods to set dependencies have been proposed, it is time to see how a sequential output from an algorithm can be parallelized. There are many ways to measure this, and a first possibility is to add moves or actions into subsets with multiple moves/actions. As moves are added to subsets, dependencies are resolved on other moves such that these moves can be added to new subsets. The list below shows subsets of moves from the current example.

- Subset 0: Move 0, Move 4
- Subset 1: Move 1, Move 5
- Subset 2: Move 2, Move 6
- Subset 3: Move 3, Move 7

Because robot behavior is complex and it is hard to calculate exactly when dependencies will be opened, two moves in the same block might not be executed at the same time. In the current example, the blocks represent what is actually going on in parallel, but what if stack 0, 1 and 2 were positioned far away from each other while 3, 4 and 5 were neighbors? Then the moves between 3, 4 and 5 would finish long before moves between 0, 1 and 2. In this simple example

the Manhattan distances in each move was equal, but for larger grids with more complex graphs that will not be the case.

Another option is to look at the longest path in the dependency graph and compare it to the sequential path. This is actually what we will do, but because the Manhattan distance is what matters, we will calculate the *minimum parallel Manhattan distance* (i.e. the lowest possible Manhattan distance if we assume unlimited robot capacity and all moves can be performed exactly when their dependencies are resolved), and compare it to the sequential total Manhattan distance.

Because the first method consider moves and the second pickup and delivery actions, they are a bit unsuitable for comparison. But the methods below should give an indicator of a total parallel Manhattan distance. The two tables below show how parallel Manhattan distance is calculated, remember that in this simple example all moves have Manhattan distance 1.

Calculating parallel Manhattan distance - moves

Assume pickups happen exactly when the move it depends on is finished. We will also assume that the robots can get to the pickup stack in zero time, i.e. robots are always located at pickup stacks. Then a move's parallel distance is the largest distance of its two dependencies plus the distance of the move. In the current example this would give a total parallel Manhattan distance of 4.

M	Dep.Dist. 1	Dep.Dist 2	Move dist.
0	0	0	1
1	**1**	0	2
2	1	**2**	3
3	2	**3**	4
4	0	0	1
5	**1**	0	2
6	1	**2**	3
7	2	**3**	4

Table 3.5: Parallel Manhattan distance with move dependencies

Calculating parallel Manhattan distance - actions

We will keep the same assumptions made for parallel Manhattan distance for moves, but now the move's parallel distance is calculated differently:

- A **Pickup action's** parallel distance is equal to the largest of its dependencies distances.
- A **Delivery action's** distance is equal to the largest of its dependencies distance plus the move's own distance.

This small modification makes sure moves dependent on pickups can be started as soon as the pickup dependency is started. Of course, this is a huge simplification, and an actual robot needs to get to pick up point before the move can start. However, compared to move dependencies, it is more correct than waiting for a whole move dependency to open. Table 3.6 shows parallel move distances.

M	Dep.Dist. 1	Dep.Dist 2	Move dist.
0	0	0	1
1	0	0	1
2	**1**	0	2
3	1	**2**	3
4	0	0	1
5	0	0	1
6	**1**	0	2
7	1	**2**	3

Table 3.6: Parallel Manhattan distance with P/D dependencies

Section 4.4 contains results that show how the sequential outputs from algorithms can be parallelized.

Chapter 4

Results

The current chapter will provide the results on how the different algorithms perform. All simulations were run on a MacBook Air 1.7 GHz Intel Core i7 with 8 GB 1600 MHz DDR3 RAM. Whether a machine with this amount of computational power is realistic in an actual storage system is not discussed, but for comparing different algorithms it is sufficient. All results can be related to what has been discussed in chapter 3. Table 4.1 provides an overview over how sections from chapter 3 and chapter 4 are related to each other.

What	Method	Results
Exhaustive search	3.2.2	4.1
DigFill V1, V2 and V3	3.5	4.2
DigFill select digStack	3.5.4	4.2.8
DigFill with greedy search	3.6	4.3
Parallelization	3.7	4.4

Table 4.1: What results belong to which section in Chapter 3

The results will be presented in the same order as the methods are presented in chapter 3. In section 4.1 results that an exhaustive search is infeasible for realistic grids is shown by some simple examples. Then the following experiments use realistic grids from table 4.2. The L1 and L2 objects are distributed such that each level in the grid contains approximately the same number of L1, L2 and non-prioritized objects.

Grid dimensions	*# of objects*	# of L2 objects	# of L1 objects
20x20x10	3640	1200	0
20x40x10	7280	2400	0
20x40x16	12080	3000	0
70x50x16	24866	4350	650
70x80x16	26522	4796	4796

Table 4.2: Grid dimensions and number of objects

4.1 Exhaustive search

Back in section 3.2, number of permutations and growth of an exhaustive search was discussed. Here, a few results shows that realistic grids cannot use an exhausting search to solve the reorganization. The tables 4.3, 4.4 and graphs in figures 4.1, 4.3 show results on how the state space grows when running the algorithm on two different grids. Remember that the exhaustive search uses an iterative approach and the leftmost column in the table is how many moves that are allowed in the current iteration.

4.1.1 Small grid

The small grid is the same grid used to illustrate the growth of an exhaustive search in figure 3.9. The grid has four stacks, each with a maximum capacity of four objects. There are four columns (Small ID, Small VAL, Small ID infinite, Small VAL infinite) showing results from the simulations.

As explained in section 3.2.1 combinations can be viewed by either its id (*ID*) or value (*VAL*). ID means that we consider the different objects as unique objects although they may have the same priority. VAL is more realistic and compares the states based on the objects priorities. However, if we had more than three different priorities the ID-states gets more realistic. The *infinite* grids are grids where all stacks have infinite maximum capacity and they are used to compare the size of state space to the function $(n \times (n-1))^m$, which is how the state space would grow if states was not visited again. An optimal solution in terms of number of moves is found after nine moves (last number in the leftmost column in 4.3).

	Small ID	Small VAL	Small ID infinite	Small VAL infinite	(n*(n-1))^m
1	13	13	13	13	12
2	73	53	109	83	144
3	271	122	763	368	1728
4	907	236	4555	1268	20736
5	2851	428	23851	3600	248832
6	8342	746	110906	9083	2985984
7	24578	1405	464642	19597	35831808
8	68440	2455	1765816	37694	429981696
9	186770	3903	OutOfMemoryError	65192	5159780352

Table 4.3: Shows how the state space grows as more moves are allowed

The graph in figure 4.1 shows how the state space grows faster when using IDs instead of priority values. The finite VAL graph (red) does not grow as rapid as the others and the optimal solution is found without searching that many states. By comparing the finite and infinite runs, it is clear that the high fill rate are limiting the growth factor quite a bit. When using the priorities to compare in a grid with a finite capacity, 3903 states was visited after only nine moves.

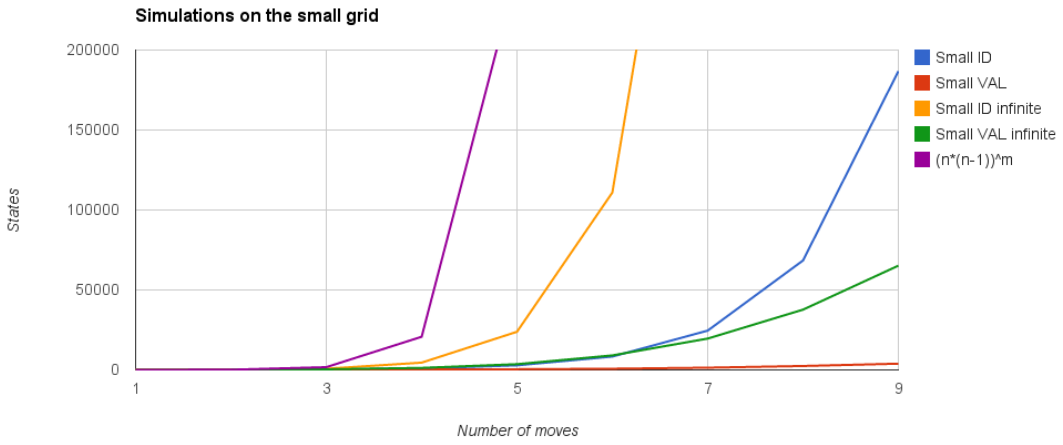

Figure 4.1: How the state spaces grow on different grids

4.1.2 Medium grid

The medium grid is slightly larger, but adding two more stacks does not make it anywhere near realistic. Figure 4.2 shows the grid used in the upcoming simulations. The maximum capacity of each stack is four objects.

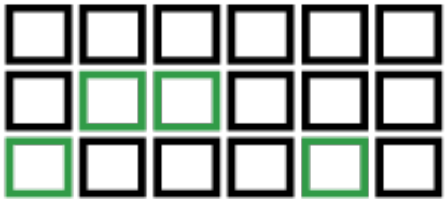

Figure 4.2: The medium sized grid

Table 4.4 shows how the state space grows. For the medium sized grid, the optimal solution was found after 13 moves. Increasing the search space by using IDs and removing the capacity constraint shows that we do not get anywhere near the 13th iteration. However, for the medium sized grid, a brute force search will still be able to find the optimal solution with the three different priority types and a finite stack capacity. Looking at the graph, the red line representing the realistic medium graph with priorities used to compare states and a maximum capacity grows much slower than the three other graphs.

	Medium ID	Medium VAL	Medium VAL infinite	(n*(n-1))^m
1	31	31	31	30
2	451	202	292	900
3	4366	752	2142	27000
4	31862	2100	12029	810000
5	194042	5250	50969	24300000
6	1079660	11819	183306	729000000
7	OutOfMemoryError	25063	551895	21870000000
8		47758	1442164	656100000000
9		87141	OutOfMemoryError	
10		147859		
11		233916		
12		348701		
13		489807		

Table 4.4: Table showing how the state space grows as more moves are allowed

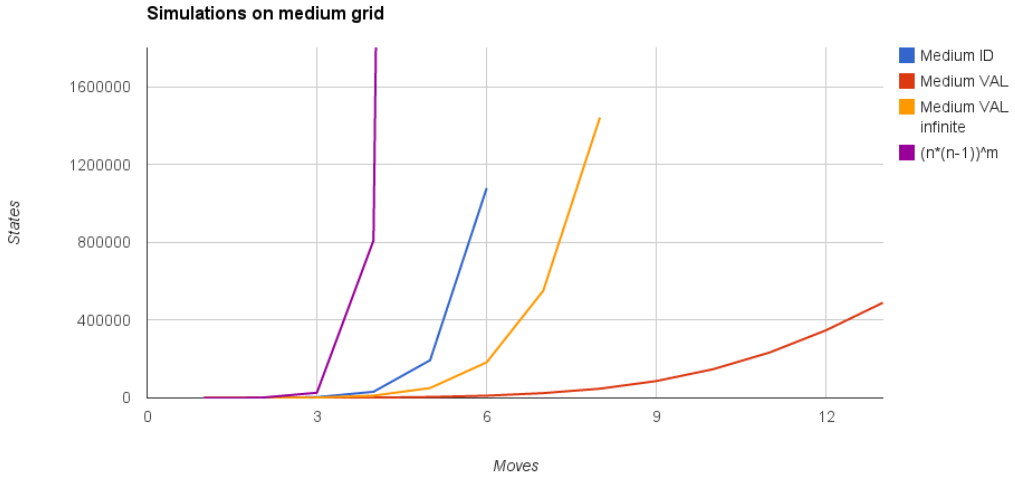

Figure 4.3: Graph showing how the state spaces grow on different grids

4.1.3 Summary

So far both grids were solvable with the brute force algorithm. It is clear that, due to only three priority types and a maximum stack capacity, the state spaces do not grow as rapid as they would if each object had its own unique priority or if the stacks had infinite capacity. However, so far the different grids have not been compared against each other.

Now, let us introduce two more grids. The first one is a *large* grid that has 10 stacks with same fill rate and capacity as the small and medium sized grids. The second is a *huge* grid with 15 stacks with the same fill rate and capacity. Table 4.5 and the graph in figure 4.4 show how the state spaces grow for each grid.

	Small VAL	Medium VAL	Large VAL	Huge VAL
1	13	31	91	211
2	53	202	1784	9949
3	122	752	16082	201077
4	236	2100	93050	
5	428	5250	389180	
6	746	11819		
7	1405	25063		
8	2455	47758		
9	3903	87141		
10		147859		
11		233916		
12		348701		
13		489807		
	~ 1000ms	~ 3 mins 30 s	Stopped 10 mins	Stopped 10 mins

Table 4.5: Table showing how the different grids search spaces increase

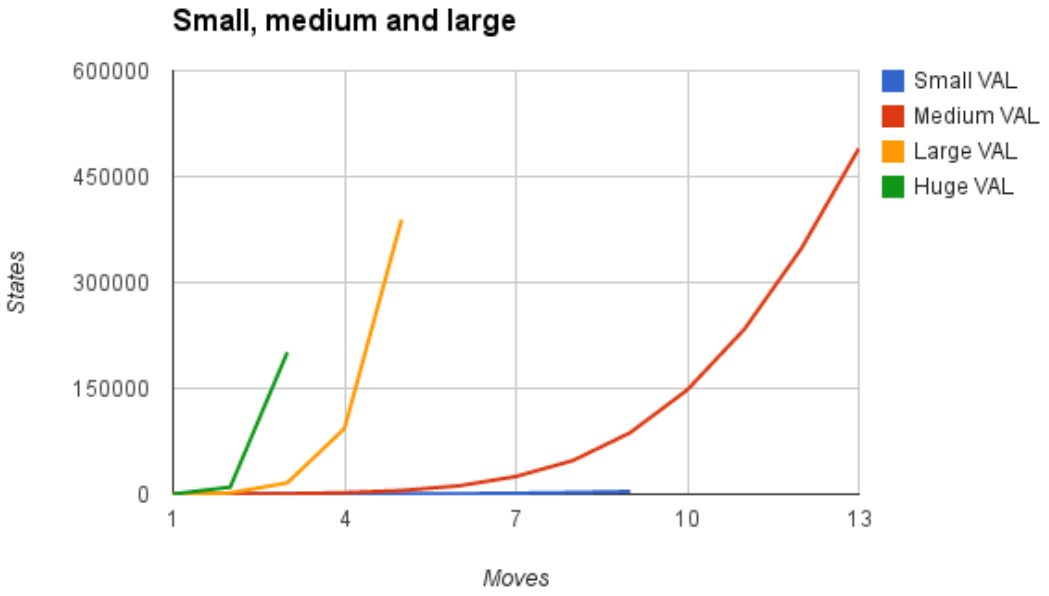

Figure 4.4: Graph comparing the grids

The large and huge grids do not get anywhere near a solution, and perhaps

with a more powerful computer these simulations would be able to finish. The optimal number of moves is unknown because we do not have an algorithm that performs a complete search at this point. Nevertheless, the basis of comparison is good enough. These grids are not even near a realistic size, but still the large and huge grids generate hundreds of thousands states in a few moves. Considering a realistic grid with $20 \times 20 = 400$ stacks that will be used in simulations later, it would generate $400 \times 399 = 159.000$ states in one move.

A last, and very important point, is the increase in runtime. While a solution for the small grid can be found in one second, it takes 3.5 minutes for the medium sized grid. Keep in mind that the medium sized grid is only two stacks (or six objects) larger than the small one. It is clear that when working with really huge grids with around 100.000 objects, a brute force algorithm cannot deal with this problem.

4.2 DigFill

All results have so far been measured and compared by the number of moves. Moves has been a simple as well as suitable indicator in the initial experiments with exhaustive searches. As we start to work with larger grids and more qualified algorithms, it is important to compare the results based on how the solution will perform in an actual, physical grid. As discussed in 3.1.3, there are many ways to measure optimality of solutions, but this thesis will focus on Manhattan distance and number of moves.

Choosing digStack, unloading objects and selecting fill objects are functions that are subject to randomness. As the algorithm repeats dig-fill cycles, the grid will eventually be solved. However, a given solution is not guaranteed to be optimal, because the DigFill search is not complete. Therefore, by running the algorithm over multiple iterations, it is possible to dilute the effect of randomness and save the best answer. In this experiment, the three different versions of DigFill from section 3.5.5 are run over 20 iterations on each grid in table 4.2. An iteration is one run of DigFill that returns a solution, so after 20 iterations we will have 20 comparable solutions.

The following sections contain results from simulations with the three different versions (see 3.5.5) on grids of various size. The table in each section contains the best, worst, average and standard deviation for moves and Manhattan distance over the 20 iterations. The bottom cells in these tables contain average runtime per iteration. A percentage change from the previous version is provided in the rightmost columns. After the table, the three following graphs compare number of moves, Manhattan distance and runtime from runs on the current grid.

The five following result sections will provide figures only and a comparison

will follow after these five sections. Then three different stack selection strategies (see section 3.5.4) are tested out, DigFill version 3 is run on two enormous grids, and in the end of this section a summary of all results can be found.

4.2.1 20x20x10

20x20x10	Moves				
	Best	*Worst*	*Average*	*Std.dev*	Average % reduction
V1	12333	17087	14712	28.37	-
V2	6119	8211	6841	20.57	53.50%
V3	3198	3518	3334	8.72	51.26%

	Manhattan				
	Best	*Worst*	*Average*	*Std. dev*	Average % reduction
V1	174997	235840	205260	102.07	-
V2	91005	118973	100529	75.62	51.02%
V3	44793	50636	46898	35.86	53.35%

	Runtimes	
	Iteration (ms)	% change
V1	541	
V2	548	-1.35%
V3	805	-46.89%

Figure 4.5: Table with moves, Manhattan distance and runtime in milliseconds

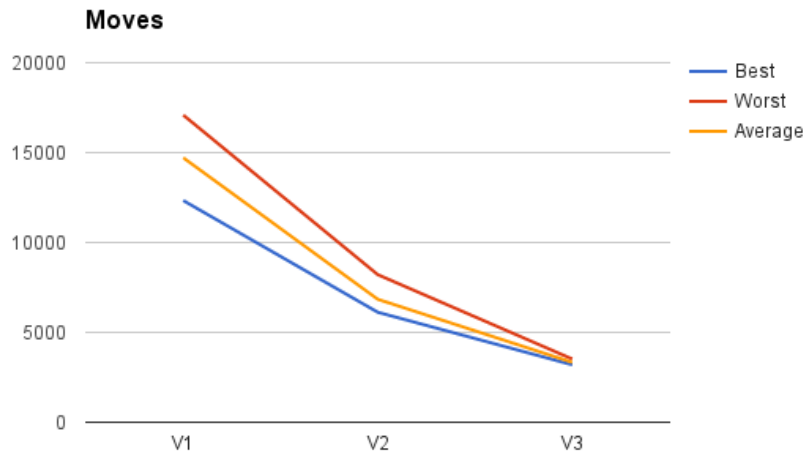

Figure 4.6: Best, worst and average moves

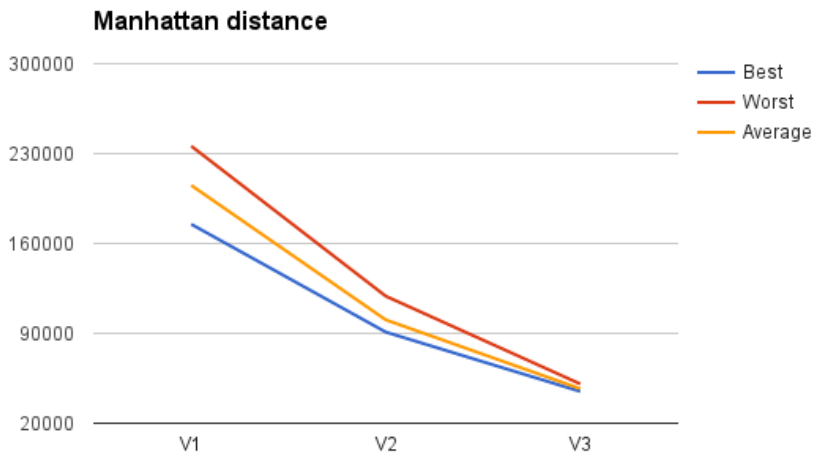

Figure 4.7: Best worst and average Manhattan distance

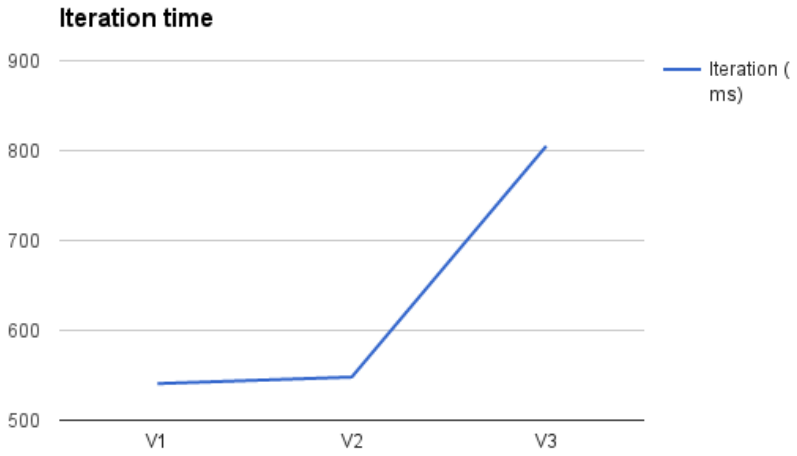

Figure 4.8: Runtimes

4.2.2 20x40x10

20x40x10	Moves				
	Best	*Worst*	*Average*	*Std.dev*	Average % reduction
V1	28729	38526	33624	41.38	-
V2	13337	17506	15088	29.50	55.13%
V3	6249	6833	6466	12.0	57.14%

	Manhattan				
	Best	*Worst*	*Average*	*Std. dev*	Average % reduction
V1	617972	812532	715498	183.41	-
V2	303195	386832	339479	131.59	52.55%
V3	128958	151509	137488	72.94	59.50%

	Runtimes	
	Iteration (ms)	% change
V1	2092	
V2	2094	-0.10%
V3	2660	-27.05%

Figure 4.9: Table with moves, Manhattan distance and runtime in milliseconds

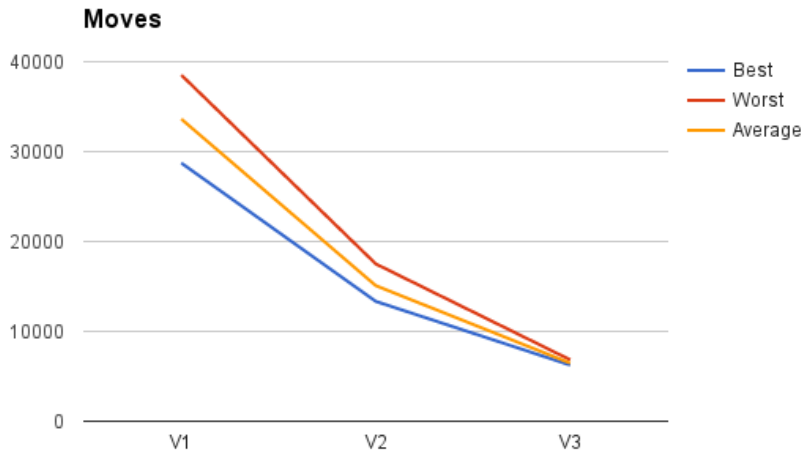

Figure 4.10: Best, worst and average moves

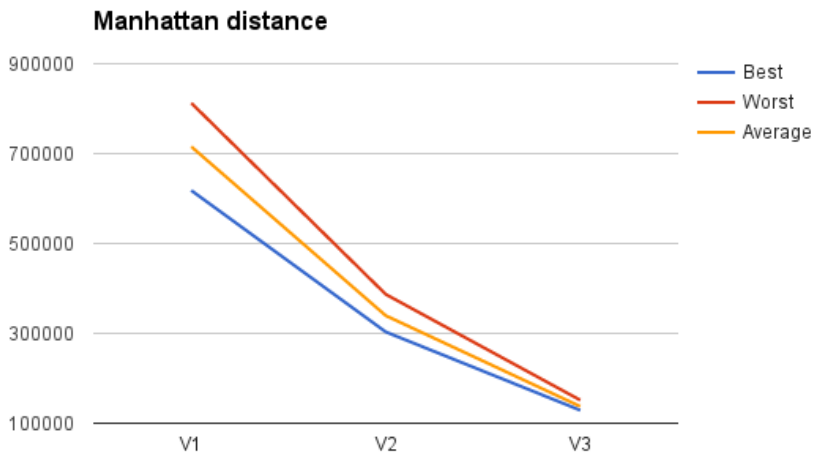

Figure 4.11: Best worst and average Manhattan distance

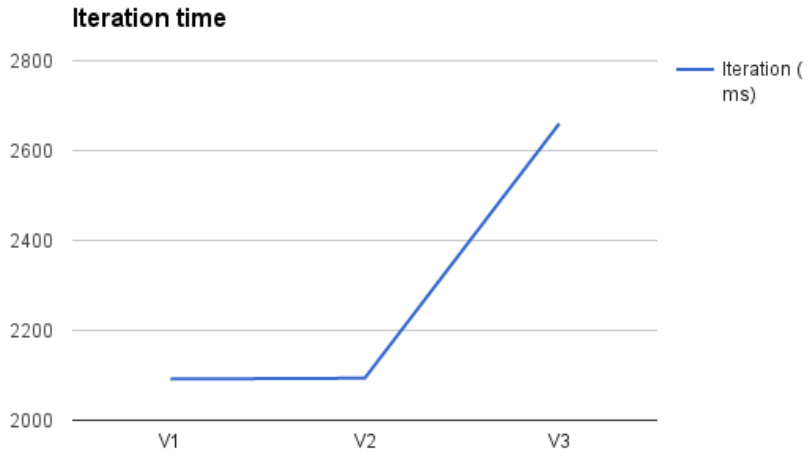

Figure 4.12: Runtimes

4.2.3 20x40x16

20x40x16	Moves				
	Best	*Worst*	*Average*	*Std.dev*	Average % reduction
V1	41533	51541	45770	45.98	-
V2	19983	25135	21881	35.38	52.19%
V3	10426	12269	10910	20.30	50.14%

	Manhattan				
	Best	*Worst*	*Average*	*Std. dev*	Average % reduction
V1	915296	1117713	1004713	204.70	-
V2	470539	579521	510532	157.74	49.19%
V3	213722	273473	231189	117.56	54.72%

	Runtimes	
	Iteration (ms)	% change
V1	2935	
V2	3209	-9.34%
V3	4372	-36.24%

Figure 4.13: Table with moves, Manhattan distance and runtime in milliseconds

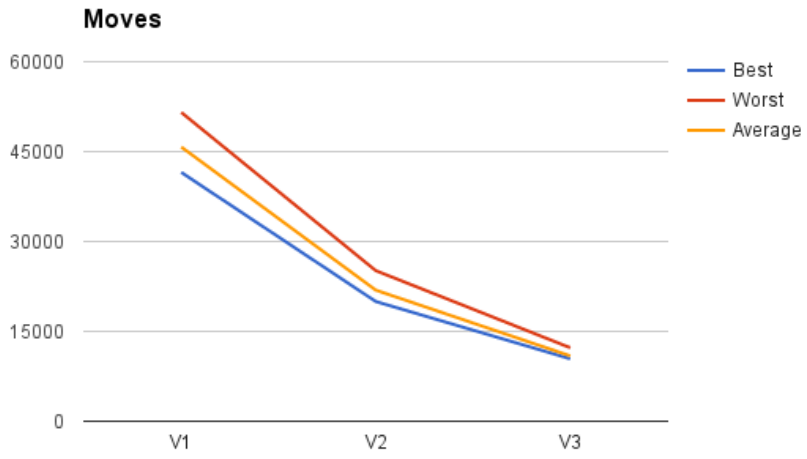

Figure 4.14: Best, worst and average moves

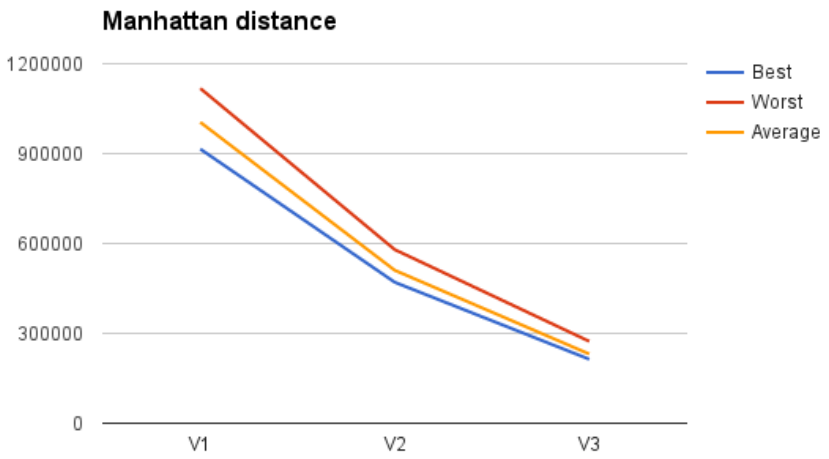

Figure 4.15: Best worst and average Manhattan distance

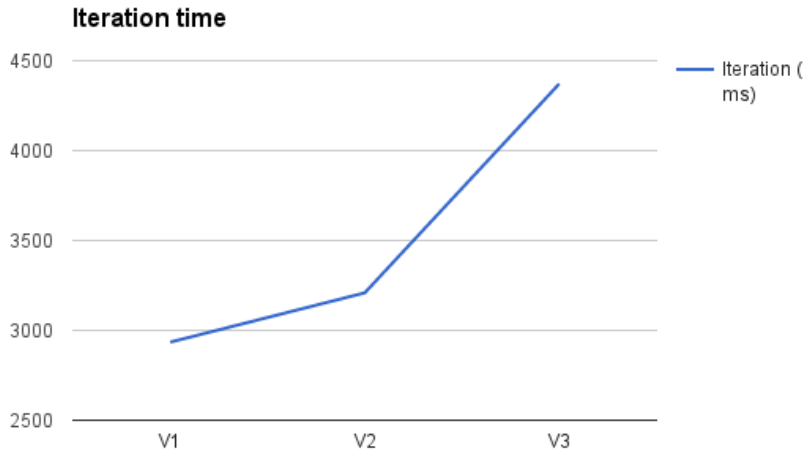

Figure 4.16: Runtimes

4.2.4 70x50x16

70x50x16	Moves				
	Best	*Worst*	*Average*	*Std.dev*	Average % reduction
V1	146840	188328	165902	96.03	-
V2	44104	57746	48903	54.08	70.52%
V3	23668	25291	24580	17.23	49.74%

	Manhattan				
	Best	*Worst*	*Average*	*Std. dev*	Average % reduction
V1	5635922	7204752	6363455	587.09	-
V2	1854584	2379581	2036932	332.21	67.99%
V3	942692	1049086	1007276	129.40	50.55%

	Runtimes	
	Iteration (ms)	% change
V1	23030	
V2	10867	52.81%
V3	29544	-171.86%

Figure 4.17: Table with moves, Manhattan distance and runtime in milliseconds

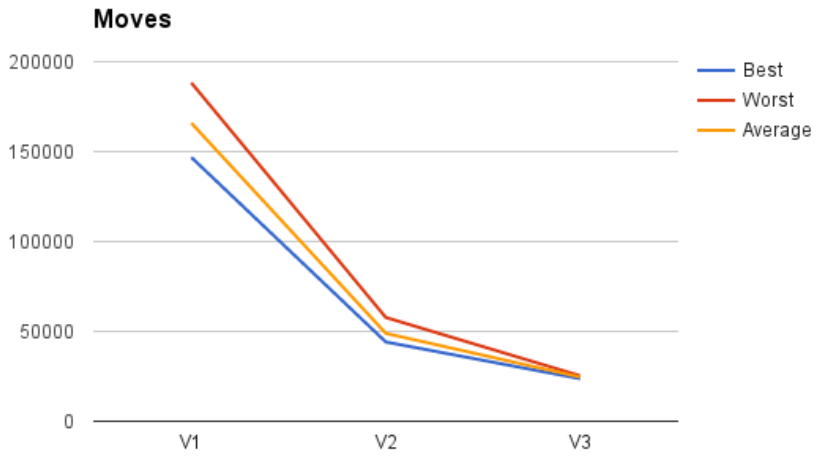

Figure 4.18: Best, worst and average moves

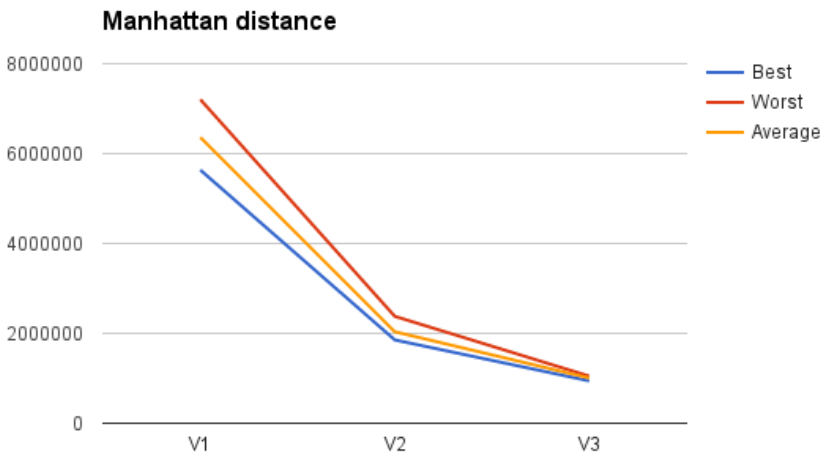

Figure 4.19: Best worst and average Manhattan distance

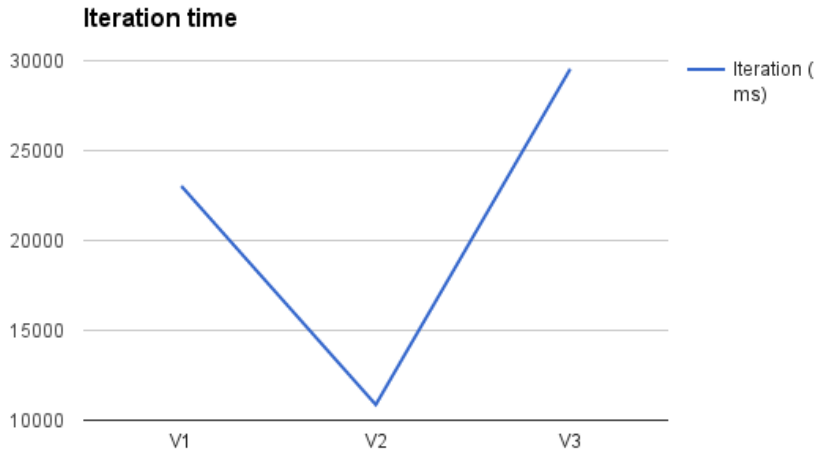

Figure 4.20: Runtimes

4.2.5 70x80x16

70x80x16	Moves				
	Best	*Worst*	*Average*	*Std.dev*	Average % reduction
V1	168947	207585	183730	94.64	-
V2	49308	62062	52378	42.85	71.49%
V3	25561	26683	26205	16.40	49.97%

	Manhattan				
	Best	*Worst*	*Average*	*Std. dev*	Average % reduction
V1	6940769	8501970	7552143	599.04	-
V2	2211599	2738850	2345036	270.31	68.95%
V3	1080862	1162332	1132100	133.82	51.72%

	Runtimes	
	Iteration (ms)	% change
V1	26064	
V2	12148	53.39%
V3	30121	-147.95%

Figure 4.21: Table with moves, Manhattan distance and runtime in milliseconds

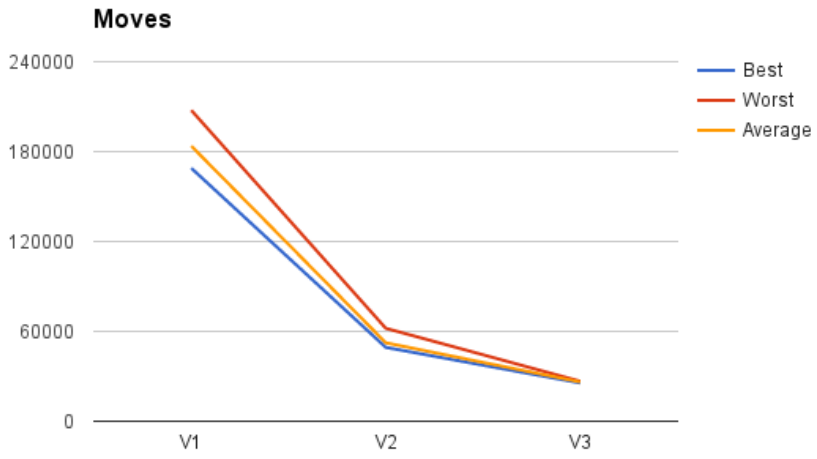

Figure 4.22: Best, worst and average moves

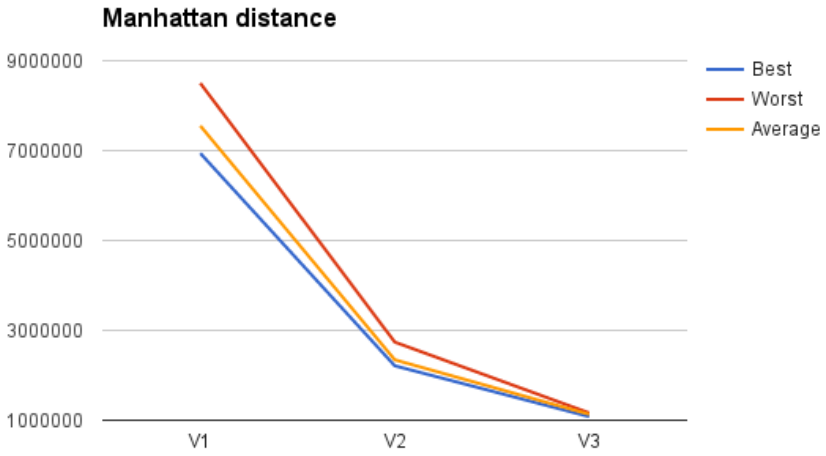

Figure 4.23: Best worst and average Manhattan distance

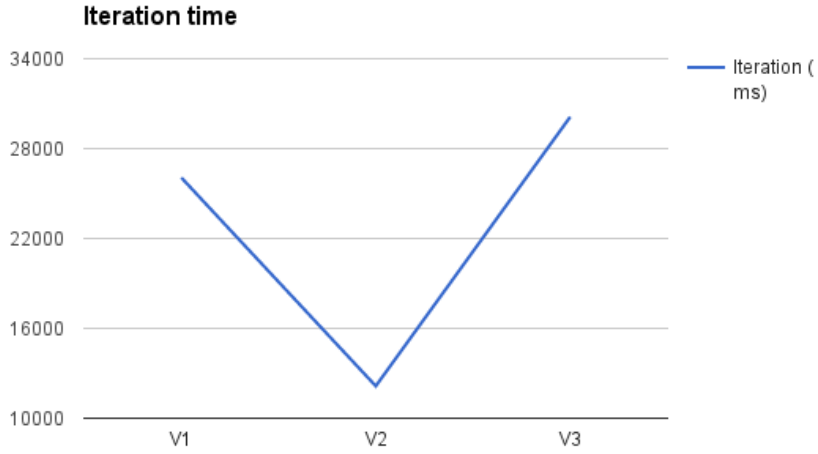

Figure 4.24: Runtimes

4.2.6 Comparing the simulations

Over 20 iterations it looks like version 2 and 3 show improvements to version 1. Moves and Manhattan distance improves a lot from version 1 to version 3. An interesting observation for version 3 is that, independent of grid size, the number of moves are getting close to the total number of objects in the grid. A closer look

into statistics on moves can be found in section 4.2.7.

For the smallest grids the average runtime graphs show an increase from version 1 to version 3. The increase is caused by the search for valid fill and unload stacks. However, for the two largest graphs this there is a drop in the graph from version 1 to 2, before the graph rises from version 2 to 3. The drop means that because the search space for those two grids are larger, they will have a greater benefit in terms of runtime from the changes made in version 2 than the smaller grids. For all five grids the runtime graphs rise from version 2 to 3. Although the runtime gets worse, the quality of the solutions improve very much from version 1 to version 3, and that is the goal. The search for fill and unload stacks are linear time searches, and can be improved by implementing a more suitable data structure.

The standard deviation decreases from version 1 to version 3. However, relative to the size of moves and Manhattan distance, the standard deviation have not changed much. Overall the standard deviation is quite low (from 0.05% to 0.2% of the average moves) and the average solution is relatively close to the best solution. Thus, the algorithms is not so affected by randomness and it is expected that each iteration gives relatively good solutions.

Keep in mind that no techniques have been used to find the nearest pickup and delivery stacks yet. Later versions will make a real effort to reduce the total Manhattan distance.

4.2.7 Move statistics

It has been mentioned more than once that a move have certain overhead. The results so far shows that number of moves were radically reduced in version 2 and 3. In this section we will look into what is actually going on, which objects are moved and how many times is each object moved.

Below are some results that show how number of moves decreases. All popped stacks are moved somewhere, so a *pop* is equivalent to a move. *Total pops* represents the total number of moves. *Popped at least once* is the number of objects moved at least once. *% popped at least once* is the percentage of total objects popped at least once. *Average times popped* is how many times a popped object is popped on average.

In all three versions, the number of objects popped at least once is almost constant. This is because the three versions of the algorithm are using the same GridConfiguration. The GridConfiguration will not allow the algorithm under any circumstances to take detours such as digging up objects in valid substacks. Just like the earlier results, this table also confirms that the number of moves decrease.

20x20x10	Total objects	Total pops	Popped at least once	% popped at least once	Avg. times popped
V1	3640	12333	2948	81%	4.18
V2	3640	6119	2866	79%	2.14
V3	3640	3198	2865	79%	1.12

20x40x10	Total objects	Total pops	Popped at least once	% popped at least once	Avg. times popped
V1	7280	28729	5920	81%	4.85
V2	7280	13337	5763	79%	2.31
V3	7280	6249	5761	79%	1.08

20x40x16	Total objects	Total pops	Popped at least once	% popped at least once	Avg. times popped
V1	12080	41533	9836	81%	4.22
V2	12080	19983	9702	80%	2.06
V3	12080	10426	9701	80%	1.07

70x50x16	Total objects	Total pops	Popped at least once	% popped at least once	Avg. times popped
V1	24866	146840	19174	77%	7.66
V2	24866	44104	18693	75%	2.36
V3	24866	23668	18694	75%	1.27

70x80x16	Total objects	Total pops	Popped at least once	% popped at least once	Avg. times popped
V1	26522	168947	20869	79%	8.10
V2	26522	49308	20366	77%	2.42
V3	26522	25561	20363	77%	1.26

Table 4.6: Popped objects statistics

In version 3, the average pops per popped object is between 1.0 and 2.0. The lowest possible number here is obviously 1.0. But this is not possible for every grid, simply because it is not necessarily possible to move all objects directly into a correct position. As an example, assume a grid with one unsolved stack and all other stacks solved. Now the solution would be to move all invalid objects up, and then fill the stack in the correct order. This would give two pops per popped bin. So, in when comparing moves, results between 1.0 and 2.0 should be a good indicator that the algorithm is performing decent. The goal should be to get this number as close to 1.0 as possible, as long as it does not affect the Manhattan distance significantly.

4.2.8 Selecting digStack

Back in section 3.5.4 there was suggested a few ways to sort the digStack. Here we will see whether the ordering of the digQueue matters or not. As the queue gets sorted, the algorithm has to follow the ordering of the objects at all times. Results from three different versions of DigFill version 3 can be found below. Manhattan distances are not compared because no techniques have yet been applied to decrease the distance. The three versions of stack selections are run on the same five grids as earlier. The next five section will again contain mostly figures and a

summary can be found in the last of these five.

20x20x10	Moves				Runtimes
	Best	*Worst*	*Average*	*Std.dev*	Iteration (ms)
Random	3198	3518	3334	8.72	805
Mismatch	3466	3579	3516	4.24	881
Fill opportunity	3199	3523	3332	8.49	903

Figure 4.25: Table comparing the three sorts

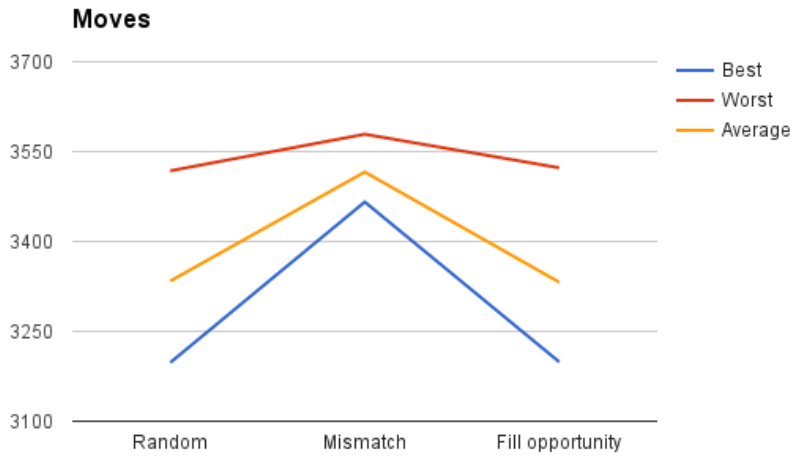

Figure 4.26: Best, worst and average moves

20x40x10	Moves				Runtimes
	Best	*Worst*	*Average*	*Std.dev*	Iteration (ms)
Random	6249	6833	6466	12.0	2660
Mismatch	7003	7264	7123	6.78	3225
Fill opportunity	6317	6959	6545	11.00	2969

Figure 4.27: Table comparing the three sorts

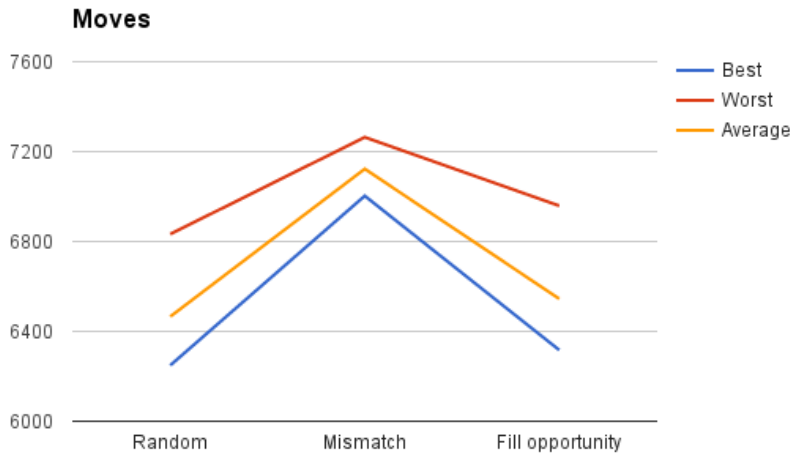

Figure 4.28: Best, worst and average moves

20x40x16	Moves				Runtimes
	Best	*Worst*	*Average*	*Std.dev*	Iteration (ms)
Random	10426	12269	10910	20.30	4372
Mismatch	12330	12581	12436	7.55	6069
Fill opportunity	10583	12261	10874	15.84	4998

Figure 4.29: Table comparing the three sorts

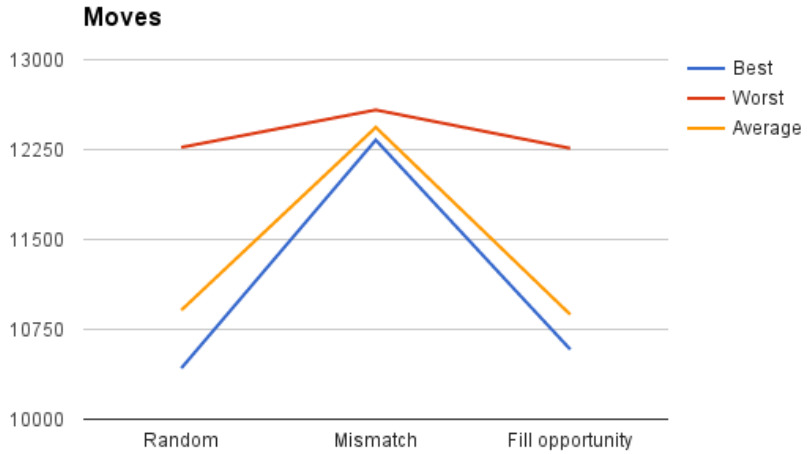

Figure 4.30: Best, worst and average moves

For the smallest grid it looks like random and fill opportunity sorting performs better than the last one. The reason fill opportunity performs so well is probably related to the small grid size. When the grid is small, the surface is also small. Hence, there are fewer fill objects to choose from. It can also been seen in the graphs that the mismatch sorting performs worse than the two other sortings.

The three smallest grids has a clear kink in their graphs, where mismatch sort performs worst and either random or fill opportunity sort is best. Not very much separate these two, so for the three first grids it is close to a tie. However, the fill opportunity sorting is really slow, which speaks in favor of choosing stacks randomly.

70x50x16	Moves				Runtimes
	Best	*Worst*	*Average*	*Std.dev*	Iteration (ms)
Random	23668	25291	24580	17.23	29544
Mismatch	24740	32432	25692	31.26	30909
Fill opportunity	34816	49127	40769	55.40	98614

Figure 4.31: Table comparing the three sorts

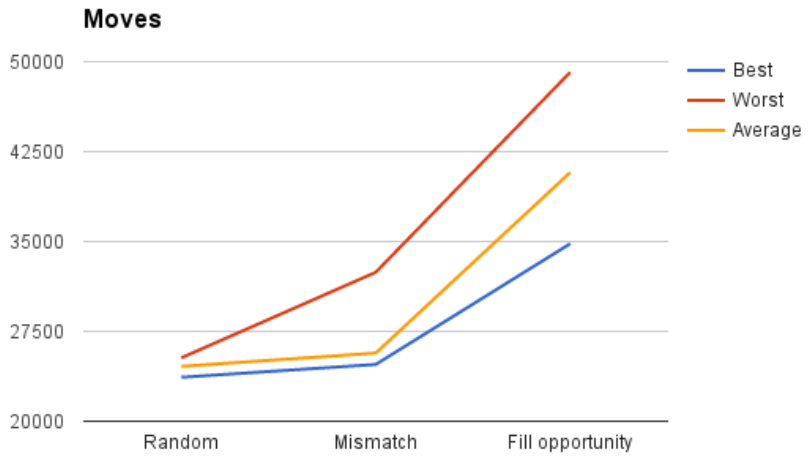

Figure 4.32: Best, worst and average moves

70x80x16	Moves				Runtimes
	Best	*Worst*	*Average*	*Std.dev*	Iteration (ms)
Random	25561	26683	26205	16.40	30121
Mismatch	28808	35416	31517	40.34	47490
Fill opportunity	34154	43342	38777	43.58	86996

Figure 4.33: Table comparing the three sorts

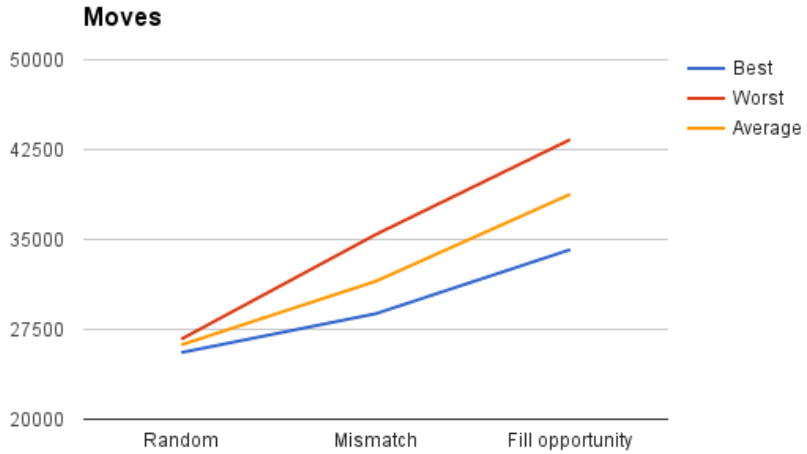

Figure 4.34: Best, worst and average moves

For the two largest grids, it is pretty clear that the random selection is best result. It is not just the best in terms of moves, but it is also the fastest all over. Summed up, the different sortings tried out here does not matter much. Perhaps there is an optimal selection strategy, but for now the random selection remains as the most stable version.

4.2.9 Two enormous grids

The grids used in the simulations have dimensions that are realistic for an actual storage. To make sure DigFill can solve larger grids within acceptable time, simulations were performed and results are found in this section. The simulations are only run with version 3 of DigFill, because the two other seem to be inferior algorithms.

Grid dimensions	# of objects	# of L2 objects	# of L1 objects
160x70x16	147111	20000	0
100x100x24	415800	20000	0

Table 4.7: Results for the two enormous grids

DigFill can handle these grids as well. Although they take longer time to solve, it is still feasible to work with grids of this size. It would have been interesting to see how the grids can be divided into zones, and solved piece by piece. As these grids get really large, the search space grows too. But the large 147.111 object grid is only 12 times the size of the 12.080 object grid. Still, solving the large one takes on average 213.137 ms per iteration, that is 151 times the 12 ms the algorithm uses per iteration on the small grid. The potential for improvement is probably really huge if we could solve split up the large grid and then solve it. These two grids have the same height so solving the 147.111 object grid in 12 parts should take more or less the same amount of time as solving the 12.080 grid 12 times. Add some extra time to merge results and it should still finish long before 213.137 ms has elapsed.

V1			Moves			
Grid dimensions	# of objects	Distributi	Best	Worst	Average	Std. dev.
160x70x16	147111		152750	161106	158702	48.78
100x100x24	415800		242870	327683	295785	181.74
			Manhattan Distance			
			Best	Worst	Average	Std. dev.
			12158506	12880050	12588604	465.81
			22511327	35524260	30700284	2280.05
			Runtime			
			Total runtime (ms)	Avg. iteration runtime (ms)		
			1065686	213137		
			1858721	371744		

Table 4.8: Results from the two large grids

4.2.10 Summary

While the exhaustive search explore the whole search space, DigFill is an algorithm using heuristics to find solutions. The search is not complete and therefore there is no proof that DigFill has found the global optimum in number of moves or Manhattan distance. However, the results so far are very promising with DigFill as a base algorithm. It solves grids of relevant size and that was the initial goal. It is important to value the work that the support structure does. Grid Configuration was created so DigFill should know where to dig and where to fill. It seems to be

doing the book keeping work very well, such that and DigFill or any other base algorithm can use it to move objects around.

Although a base algorithm could have been implemented differently, the main goal was to step by step get all objects in place. The graphs in figures 4.36 and 4.37 show how the number of unsolved stacks are decreasing as the DigFill version 3 works in some of the grids. The algorithm follows the same steps at all times and solves the grid stack by stack, thus the linear pace of the algorithm is just like expected.

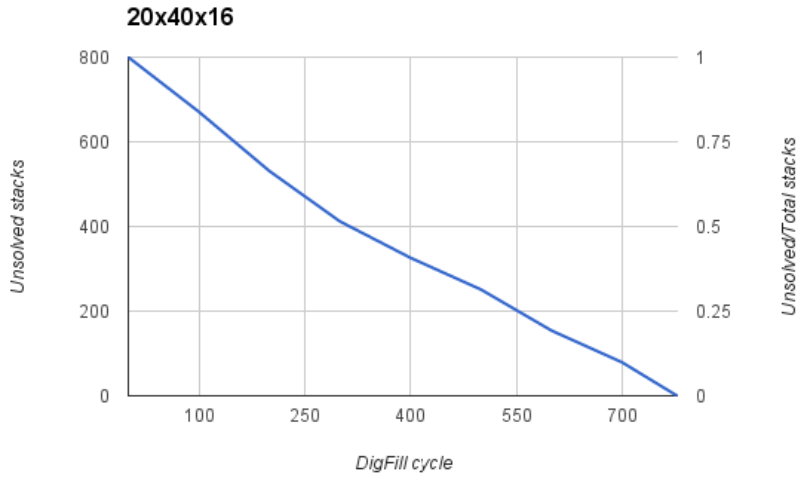

Figure 4.35: How unsolved stacks decrease over DF cycles

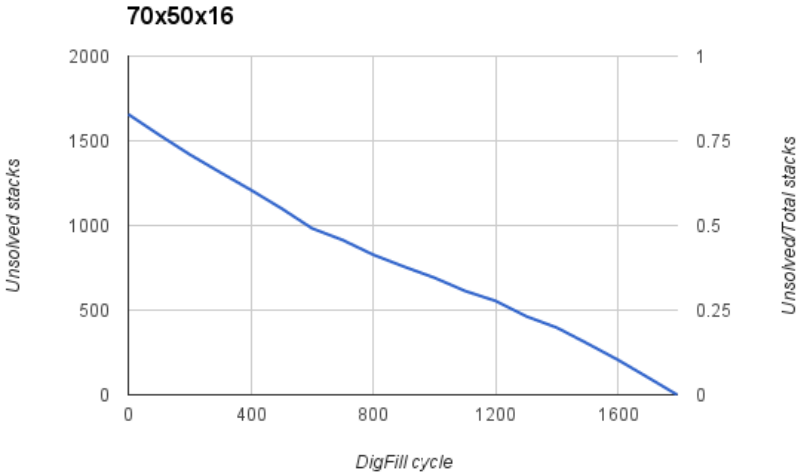

Figure 4.36: How unsolved stacks decrease over DF cycles

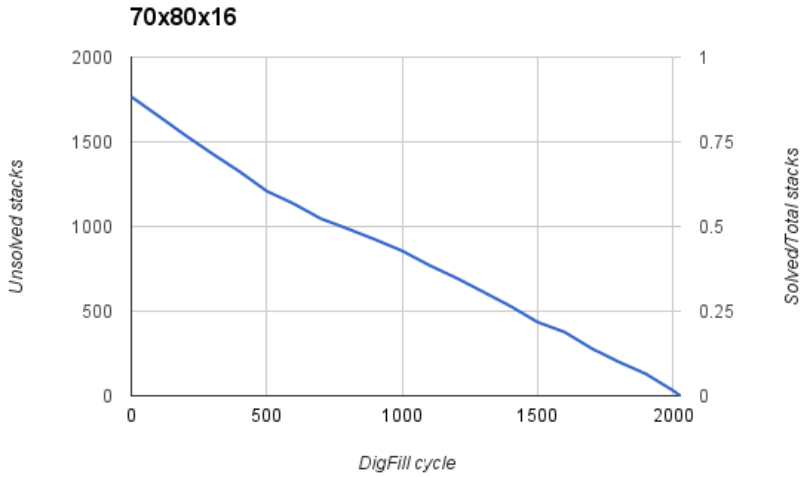

Figure 4.37: How unsolved stacks decrease over DF cycles

Improvements between version 1, 2 and 3 are shown earlier, and there is probably a lot of potential here to make the base algorithm even better. But for now, the base algorithm is good enough to simply serve as a digger and a filler. The next goal is to figure out if and how the total Manhattan distance can be reduced and if this sequential list of moves is parallelizable.

4.3 DigFill with greedy search

So far the Manhattan distances have just been numbers without anything to relate them to. It is reasonable to believe that these numbers are really bad compared to how the system performs today. A simple approach that hopefully will improve the distance a lot is greedy search. DigFill version 3 with random selection of digStack will be used as the base algorithm.

Section 3.6 described the implementation and how to perform a greedy search using DigFill as base algorithm. This section provides results on how the greedy search performs. In the same manner as previous section, results will be shown grid by grid in the next five sections. These sections contain a table comparing the standard DigFill algorithm (DigFill), results from a simulation on a real system (Real) and the greedy version of DigFill (Greedy).

The simulation on an actual grid is a real simulation with robots moving together in a grid. A robot must move to the pickup stack, pickup the object, transport it to the delivery stack and deliver the object before it can head for another object. All robots work simultaneously in the same grid and the simulation

of the real system has a very low reality gap. Unfortunately, this means that the Manhattan distance is calculated by the total distance all robots travel together, including *empty runs* i.e. runs to a pickup point or other runs without objects. This is slightly inconvenient when comparing results, but when considering both moves and distance it is possible to see some patterns. The standard deviation for the actual simulation (Real) is not reported because these results are based on only two simulations for each grid.

4.3.1 20x20x10

20x20x10	Moves			
	Best	*Worst*	*Average*	*Std.dev*
DigFill	3198	3518	3334	8.72
Real	5438	5517	5477.5	-
Greedy	5360	5629	5476	7.62

	Manhattan			
	Best	*Worst*	*Average*	*Std. dev*
DigFill	44793	50636	46898	35.86
Real	32941	33565	33253	-
Greedy	9556	11077	10473	16.49

	Runtimes
	Iteration (ms)
DigFill	805
Greedy	2241

Figure 4.38: Table with moves, Manhattan distance and runtime in milliseconds

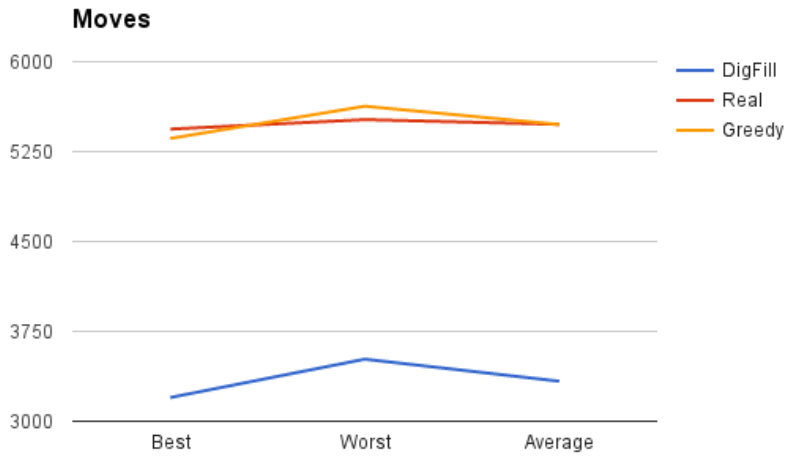

Figure 4.39: Best, worst and average moves

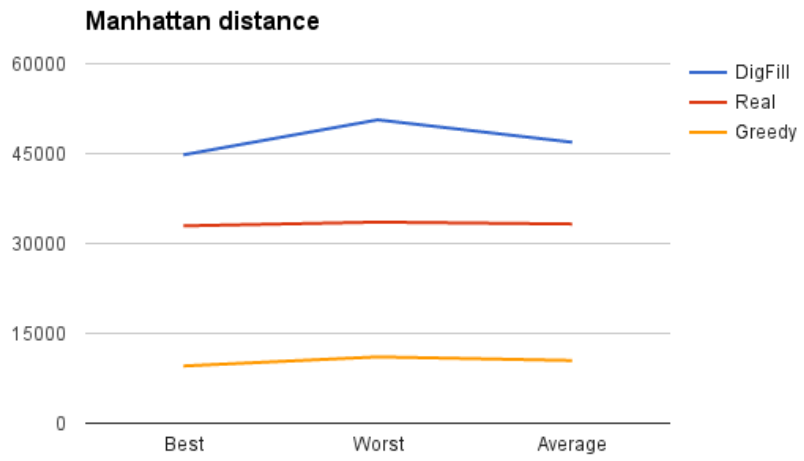

Figure 4.40: Best worst and average Manhattan distance

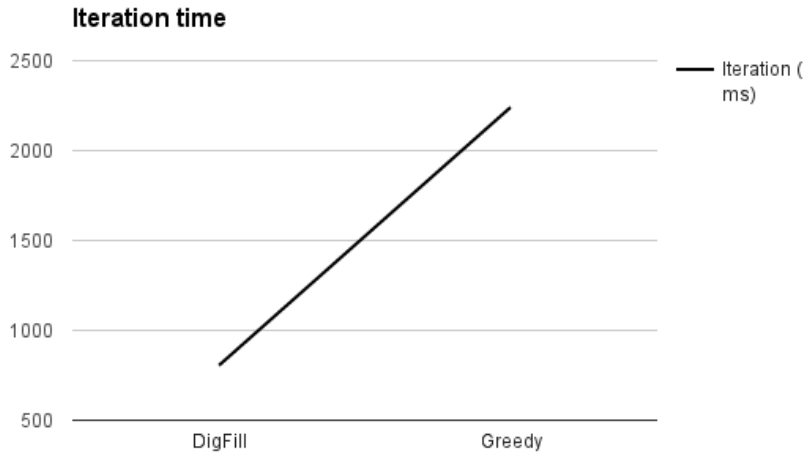

Figure 4.41: Runtimes

4.3.2 20x40x10

20x40x10	Moves			
	Best	*Worst*	*Average*	*Std.dev*
DigFill	6249	6833	6466	12.0
Real	11804	11869	11836.5	-
Greedy	10766	11333	11085	11.4

	Manhattan			
	Best	*Worst*	*Average*	*Std. dev*
DigFill	128958	151509	137488	72.94
Real	74962	76799	75880.5	-
Greedy	20618	24475	22554	30.45

	Runtimes
	Iteration (ms)
DigFill	2660
Greedy	8642

Figure 4.42: Table with moves, Manhattan distance and runtime in milliseconds

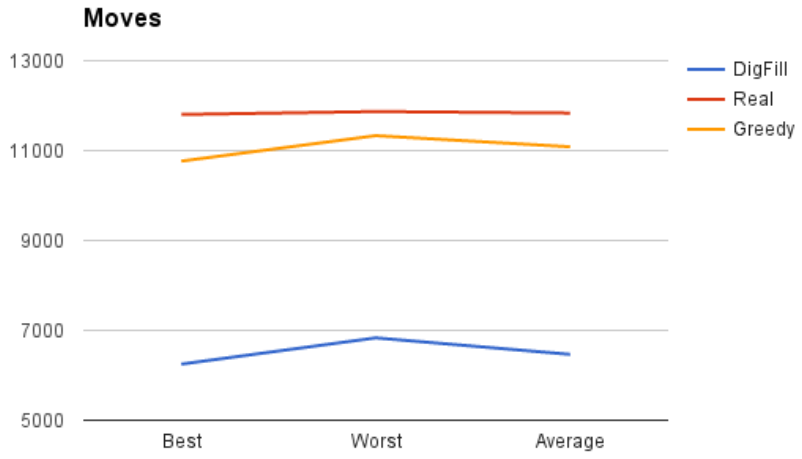

Figure 4.43: Best, worst and average moves

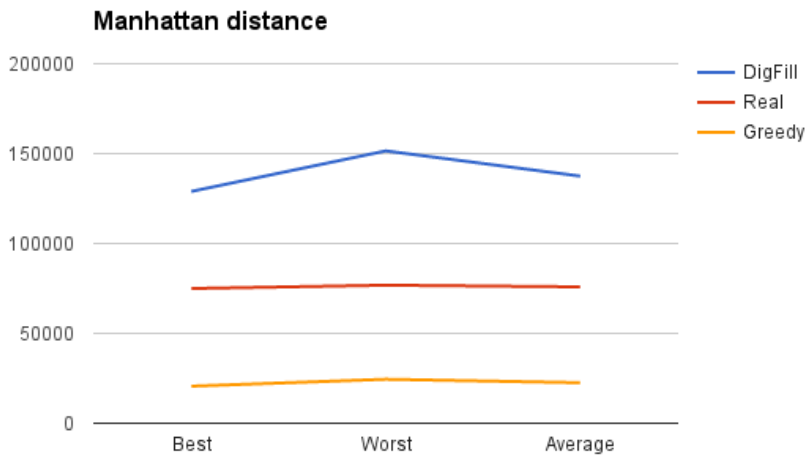

Figure 4.44: Best worst and average Manhattan distance

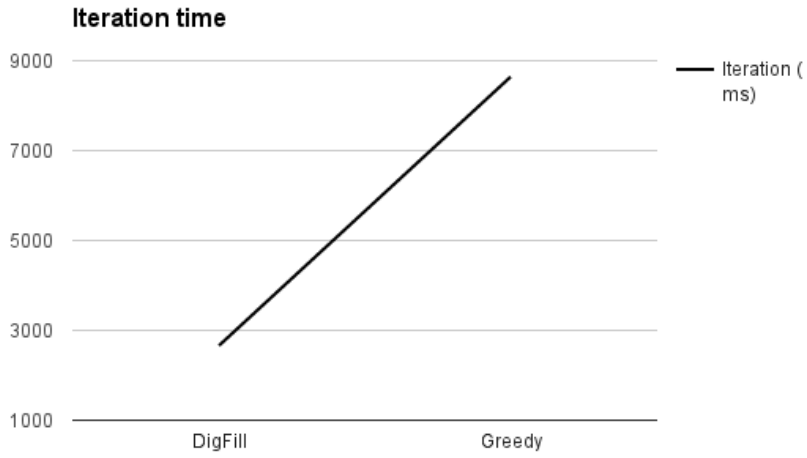

Figure 4.45: Runtimes

4.3.3 20x40x16

20x40x16	Moves			
	Best	*Worst*	*Average*	*Std.dev*
DigFill	10426	12269	10910	20.30
Real	20154	20251	20202.5	-
Greedy	18595	19386	19092	11.75

	Manhattan			
	Best	*Worst*	*Average*	*Std. dev*
DigFill	213722	273473	231189	117.56
Real	166833	170514	168673.5	-
Greedy	40296	44323	42390	31.53

	Runtimes
	Iteration (ms)
DigFill	4372
Greedy	14141

Figure 4.46: Table with moves, Manhattan distance and runtime in milliseconds

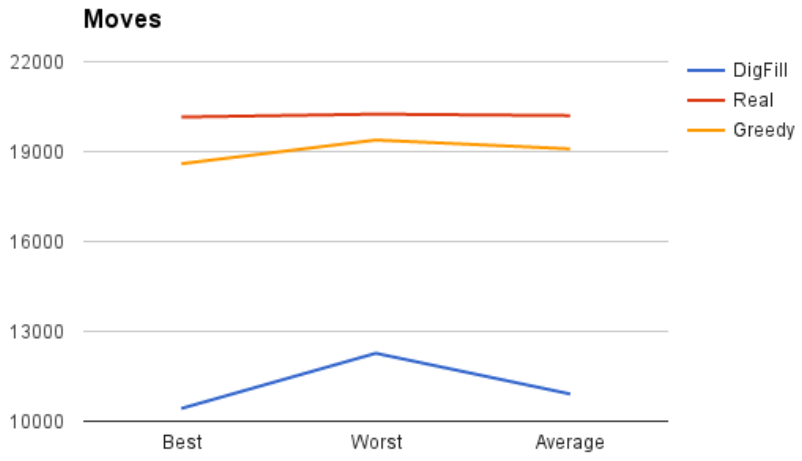

Figure 4.47: Best, worst and average moves

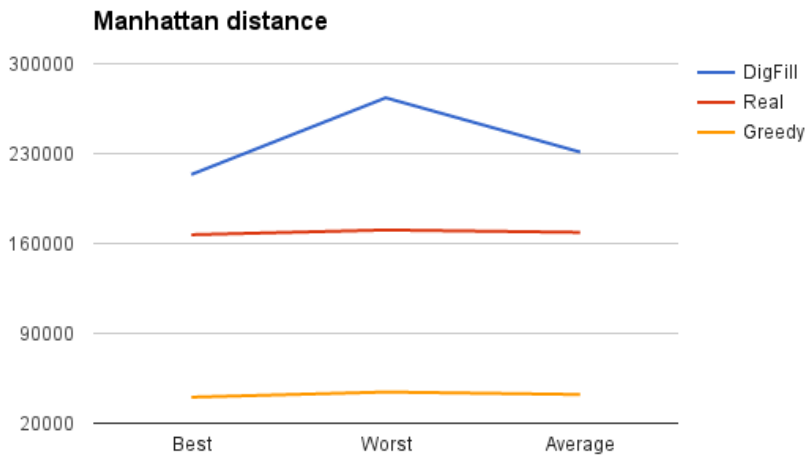

Figure 4.48: Best worst and average Manhattan distance

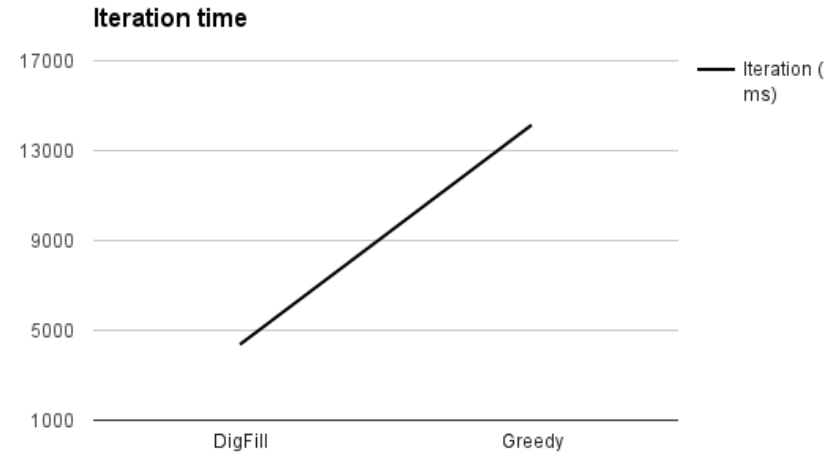

Figure 4.49: Runtimes

4.3.4 70x50x16

70x50x16	Moves			
	Best	*Worst*	*Average*	*Std.dev*
DigFill	23668	25291	24580	17.23
Real	41300	41374	41337	-
Greedy	35178	35844	35500	13.23

	Manhattan			
	Best	*Worst*	*Average*	*Std. dev*
DigFill	942692	1049086	1007276	129.40
Real	302100	308170	305135	-
Greedy	94058	108221	99775	49.30

	Runtimes
	Iteration (ms)
DigFill	29544
Greedy	62501

Figure 4.50: Table with moves, Manhattan distance and runtime in milliseconds

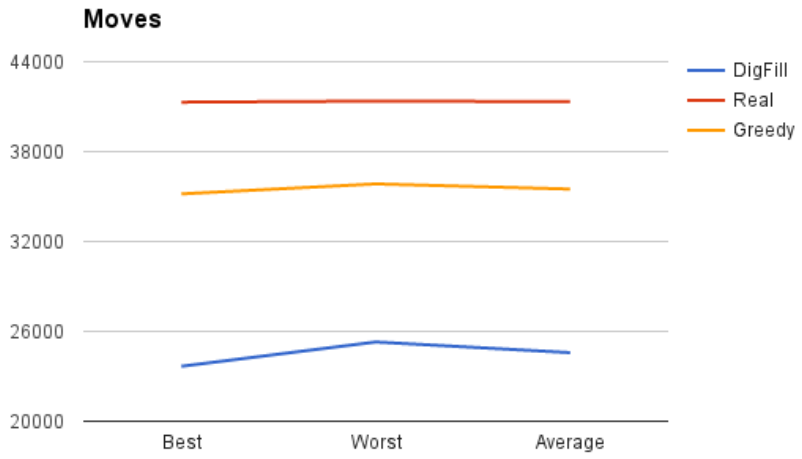

Figure 4.51: Best, worst and average moves

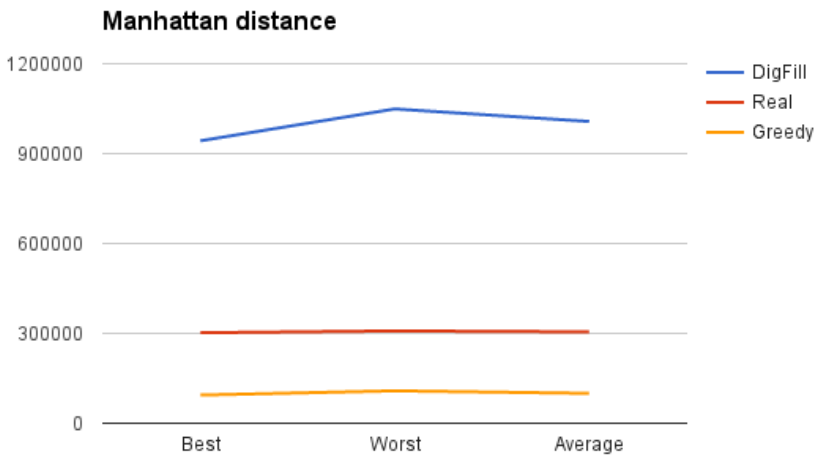

Figure 4.52: Best worst and average Manhattan distance

Figure 4.53: Runtimes

4.3.5 70x80x16

70x80x16	Moves			
	Best	*Worst*	*Average*	*Std.dev*
DigFill	25561	26683	26205	16.40
Real	46620	46772	46696	-
Greedy	38739	39484	39200	12.41

	Manhattan			
	Best	*Worst*	*Average*	*Std. dev*
DigFill	1080862	1162332	1132100	133.82
Real	341470	340086	340778	-
Greedy	101932	115770	109138	50.11

	Runtimes
	Iteration (ms)
DigFill	30121
Greedy	73659

Figure 4.54: Table with moves, Manhattan distance and runtime in milliseconds

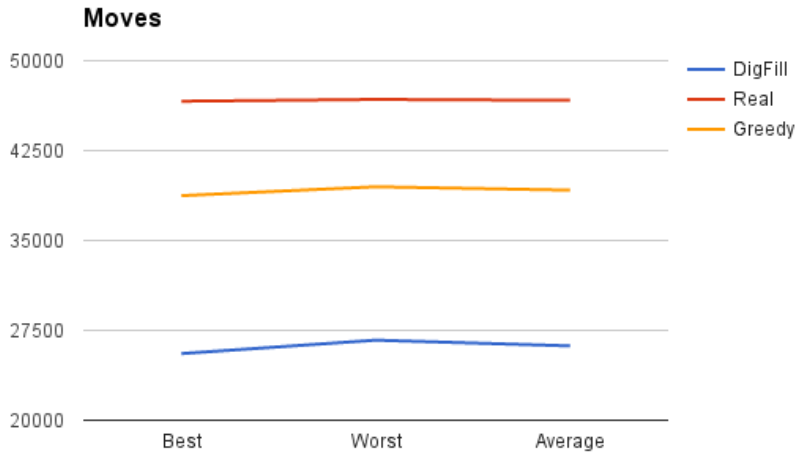

Figure 4.55: Best, worst and average moves

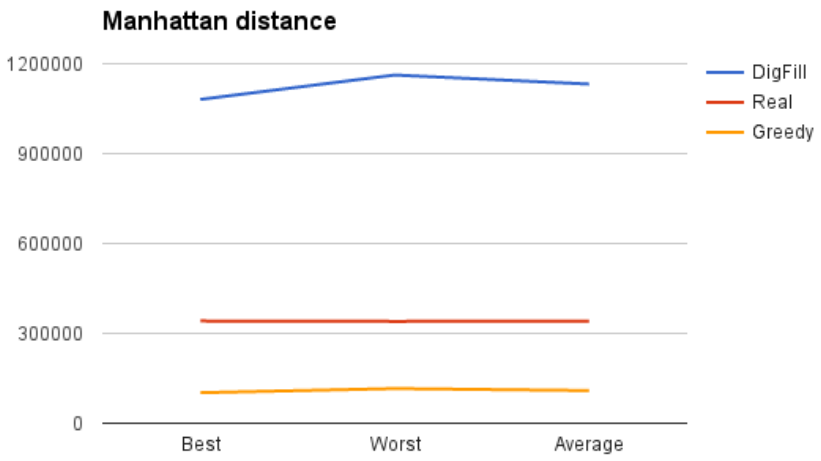

Figure 4.56: Best worst and average Manhattan distance

Figure 4.57: Runtimes

4.3.6 Observations

When a greedy search is performed, and objects are not placed all over the grid anymore, the total Manhattan distance gets much lower as expected. We see that for every grid except the smallest one, number of moves are significantly reduced compared to the simulation on today's system (Real). This is expected because

DigFill is built to put objects in a place where they do not need to be moved again, and DigFill is therefore biased towards minimizing the total number of moves.

The Manhattan distance is about a third of the total distance traveled for all robots in the actual system. As the distance of empty runs is not known, there is no solid basis to argue whether this distance is an improvement or not by just comparing the number. However, what we will do in the next section is to look at the *move-distance* trade off and then discuss what might give the best results in an actual grid if the result of the algorithm were fed into a real system.

The most fascinating observation here not how greedy performs better than standard DigFill or that the Manhattan distance is less than in the actual simulation, but rather the increased number of moves and the increase in average pop per popped stack. The number that was close to 1.0 in the standard DigFill implementation is now almost doubled for every grid when running the greedy search.

At first it seemed very strange that the greedy algorithm had such an increase in number of moves, and the first thought was that this might have something to do with the fill order if multiple objects are possible. The rule from section 3.5.3 was to always choose the lowest priority first in order if multiple priorities was available. This was based on an idea to check the lowest prioritized objects off the list or get rid of them as soon as possible. This initial assumption was wrong, and choosing the highest priority when possible performs slightly better on small grids with small L1 and L2 lists. However, the overall difference is between the two is incredibly small, so this will not be discussed further.

4.3.7 The move-distance trade off

The problem we will deal with in this section is that a the standard random DigFill is almost optimal in terms of number of moves, but gets a very long Manhattan distance which is expected due to its randomness. On the other side of the problem, we have the greedy algorithm with an increase in number of moves, but a reduced Manhattan distance.

This problem is probably caused by the fact that after a stack is dug, the stack wants to be filled up again. Picture a grid with only *one* unsolved stack. The solution to the grid would be to dig the stack and then fill it with the popped objects in the correct order. An operation that requires two moves per moved object - first pop then pushed back to the same stack.

The example above illustrates what could be the issue causing a greedy search to simply pop and then push the objects right back. However, the greedy search must follow the GridDefinition's policies and look for unload stacks that need the popped object, so the moves cannot always be done locally and the problem is

likely to have more than one cause. But for the cases like in the example, this would give an average pop per popped stack close to 2.0.

Now, even though the unloading cannot always be made locally, fill stacks probably will be found locally. Most of the time, a grid is unsolved and a multitude of different object priorities are covering the grid's surface. Thus, it is easy to find some stack nearby that peeks a wanted fill object. In many cases a fill object can be chosen among different priorities, this fact increase the probability of finding fill stacks close and augments the hypothesis that fill stacks are easier to find locally than unload stacks.

Using fill stacks to fill dug stacks, can be seen as the beginning of multiple dig steps. What DigFill actually wants is to dig a stack completely up, and then fill it with valid objects. As many digs are initiated by the fill step, these stacks (that may remain invalid) might be a delivery point for a future unloading, and the single dig was redundant.

In DigFill version 3, the dig step was greatly improved by looking for an unload stack that needs the popped object. In fact this improvement may have improved the algorithm such that the whole fill step is redundant. Just leave the dug stack open, and it will (at some point) be filled with valid objects. The fill step would never fill a stack with invalid objects anyway, and while digging we want to unload stacks in a valid position aka. filling.

The following sections contain results for each grid from the standard DigFill, the real simulation, the greedy implementation of DigFill and a greedy implementation without the fill step, Greedy Dig. After the tables and graphs are presented, a summary for the whole section 4.3 will follow.

4.3.8 20x20x10

20x20x10	Moves			
	Best	*Worst*	*Average*	*Std.dev*
DigFill	3198	3518	3334	8.72
Real	5438	5517	5477.5	-
Greedy	5360	5629	5476	7.62
Greedy Dig	2971	3141	3023	5.20

	Manhattan			
	Best	*Worst*	*Average*	*Std. dev*
DigFill	44793	50636	46898	35.86
Real	32941	33565	33253	-
Greedy	9556	11077	10473	16.49
Greedy Dig	22945	26687	24713	24.35

	Runtimes
	Iteration (ms)
DigFill	805
Greedy	2241
Greedy Dig	810

Figure 4.58: Table with moves, Manhattan distance and runtime in milliseconds

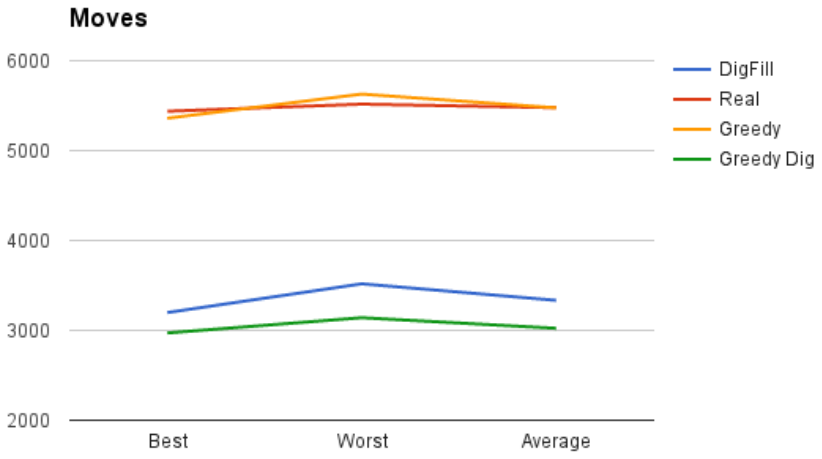

Figure 4.59: Best, worst and average moves

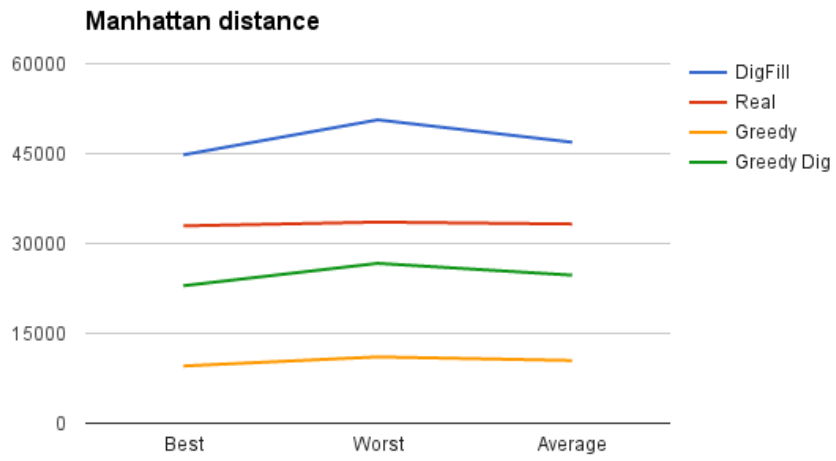

Figure 4.60: Best worst and average Manhattan distance

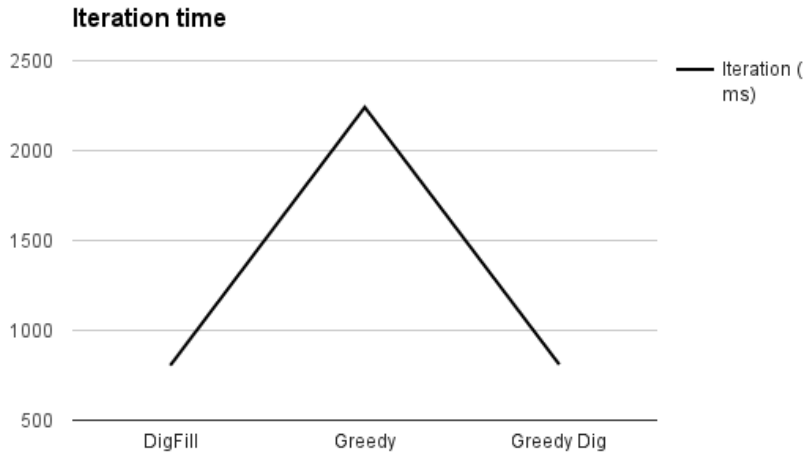

Figure 4.61: Runtimes

4.3.9 20x40x10

20x40x10	Moves			
	Best	*Worst*	*Average*	*Std.dev*
DigFill	6249	6833	6466	12.0
Real	11804	11869	11836.5	-
Greedy	10766	11333	11085	11.4
Greedy Dig	5934	6091	6012	5.2

	Manhattan			
	Best	*Worst*	*Average*	*Std. dev*
DigFill	128958	151509	137488	72.94
Real	74962	76799	75880.5	-
Greedy	20618	24475	22554	30.45
Greedy Dig	62535	69169	67077	32.02

	Runtimes
	Iteration (ms)
DigFill	2660
Greedy	8642
Greedy Dig	2768

Figure 4.62: Table with moves, Manhattan distance and runtime in milliseconds

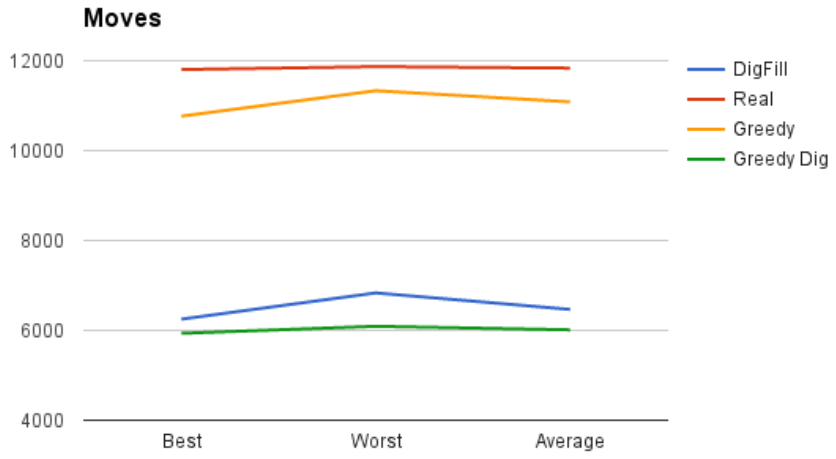

Figure 4.63: Best, worst and average moves

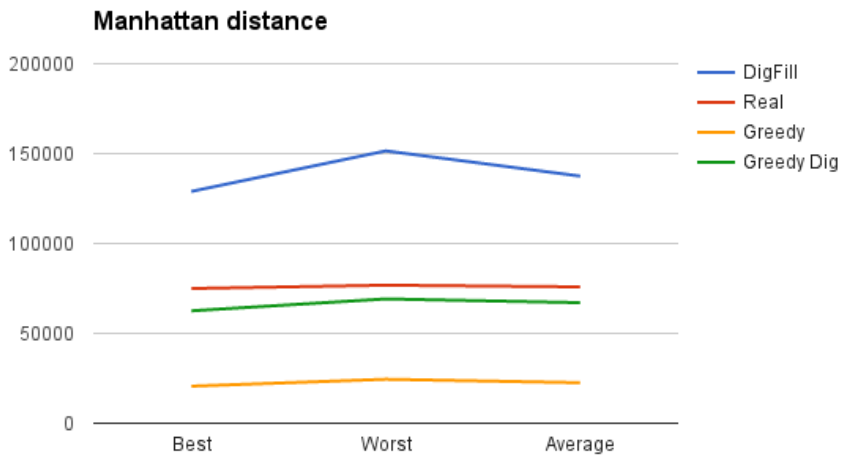

Figure 4.64: Best worst and average Manhattan distance

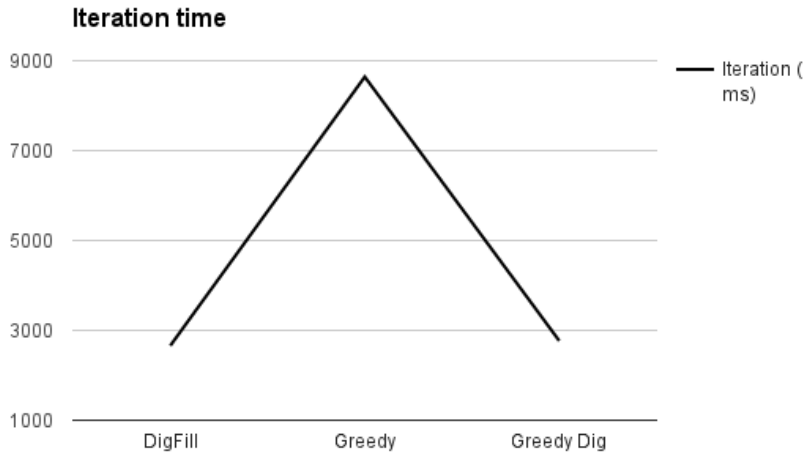

Figure 4.65: Runtimes

4.3.10 20x40x16

20x40x16	Moves			
	Best	*Worst*	*Average*	*Std.dev*
DigFill	10426	12269	10910	20.30
Real	20154	20251	20202.5	-
Greedy	18595	19386	19092	11.75
Greedy Dig	9978	10186	10067	6.08

	Manhattan			
	Best	*Worst*	*Average*	*Std. dev*
DigFill	213722	273473	231189	117.56
Real	166833	170514	168673.5	-
Greedy	40296	44323	42390	31.53
Greedy Dig	118293	134809	127360	52.35

	Runtimes
	Iteration (ms)
DigFill	4372
Greedy	14141
Greedy Dig	5821

Figure 4.66: Table with moves, Manhattan distance and runtime in milliseconds

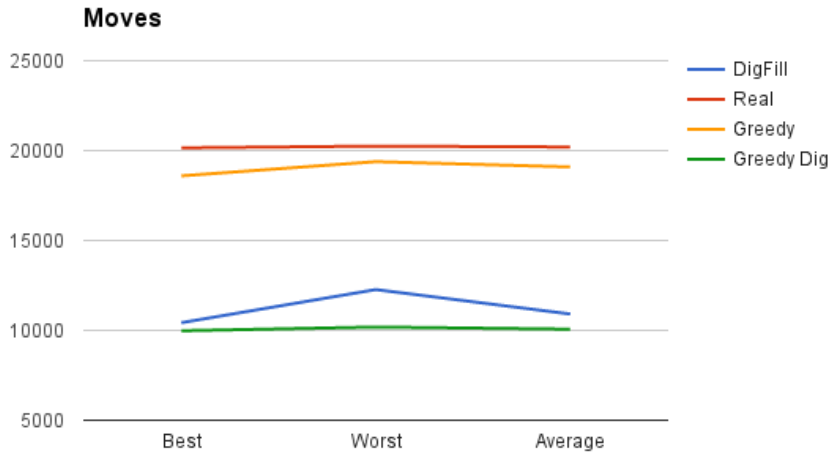

Figure 4.67: Best, worst and average moves

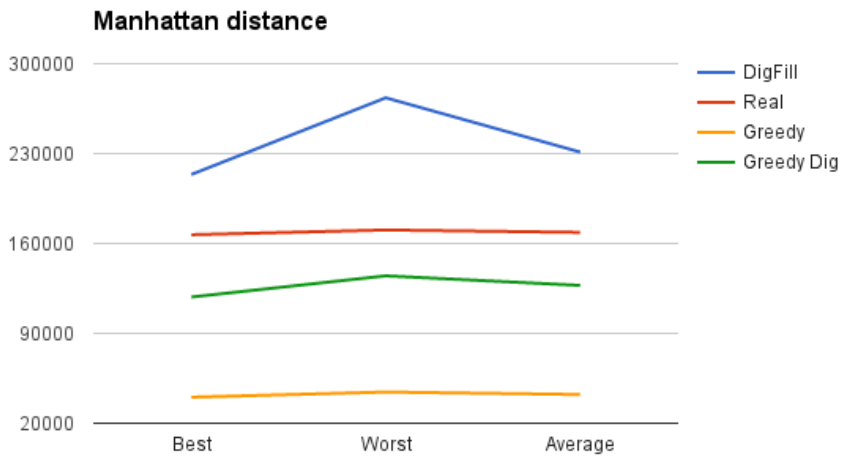

Figure 4.68: Best worst and average Manhattan distance

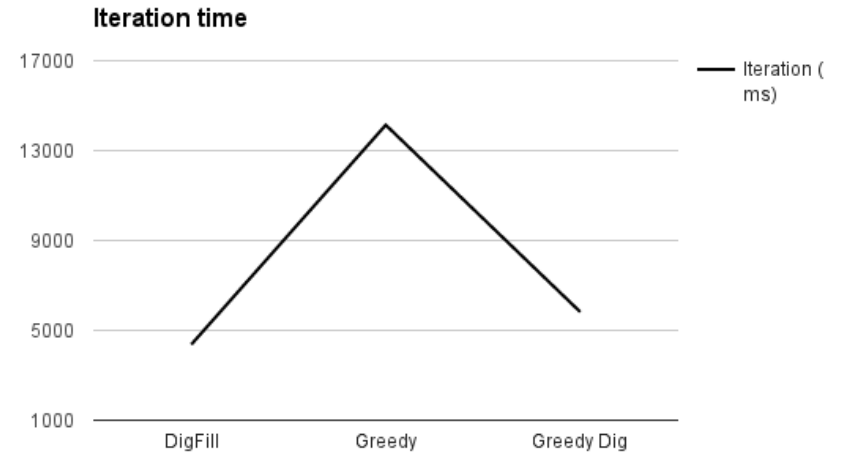

Figure 4.69: Runtimes

4.3.11 70x50x16

70x50x16	Moves			
	Best	*Worst*	*Average*	*Std.dev*
DigFill	23668	25291	24580	17.23
Real	41300	41374	41337	-
Greedy	35178	35844	35500	13.23
Greedy Dig	19065	19285	19161	6.25

	Manhattan			
	Best	*Worst*	*Average*	*Std. dev*
DigFill	942692	1049086	1007276	129.40
Real	302100	308170	305135	-
Greedy	94058	108221	99775	49.30
Greedy Dig	280727	415753	349022	167.58

	Runtimes
	Iteration (ms)
DigFill	29544
Greedy	62501
Greedy Dig	17542

Figure 4.70: Table with moves, Manhattan distance and runtime in milliseconds

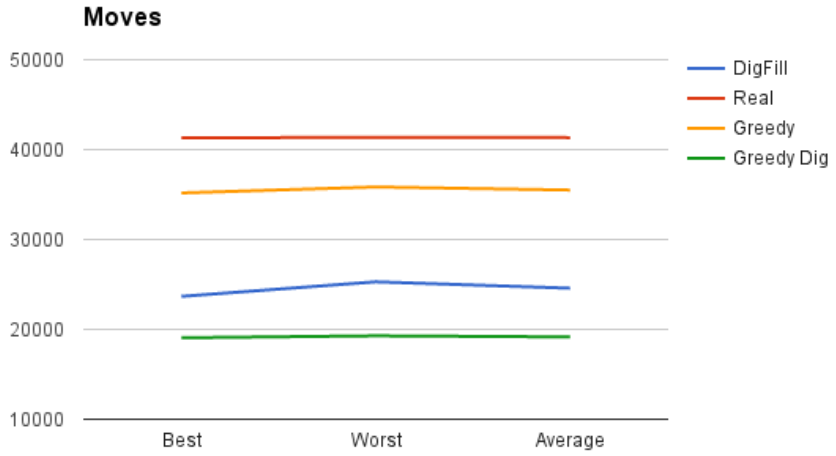

Figure 4.71: Best, worst and average moves

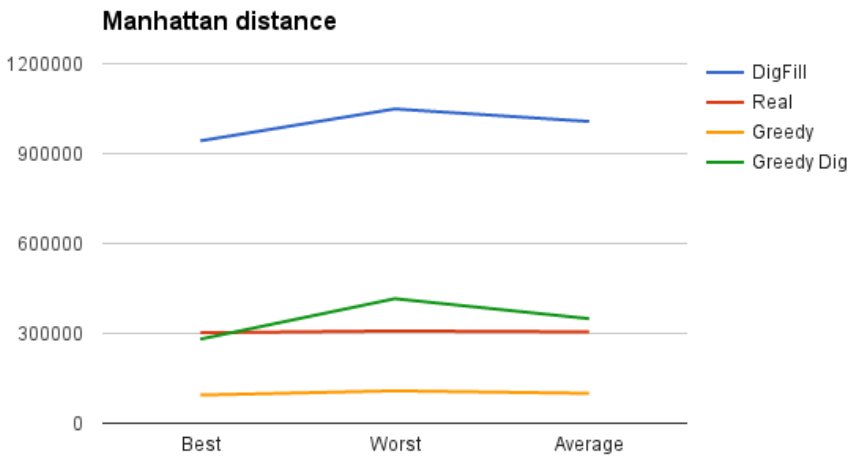

Figure 4.72: Best worst and average Manhattan distance

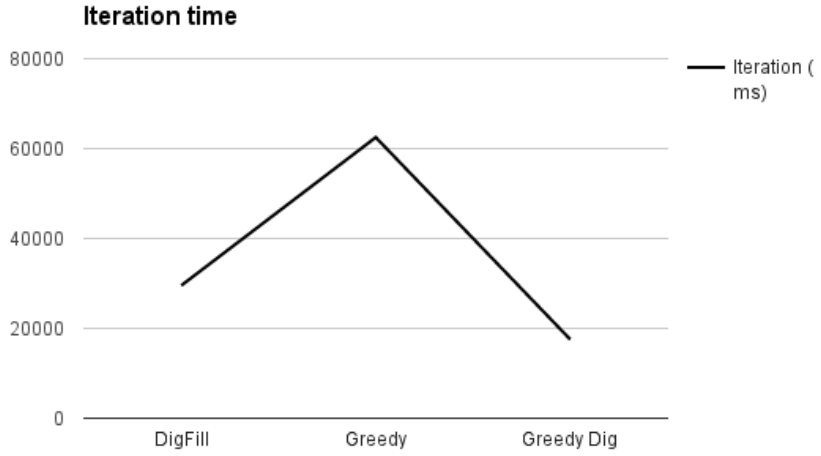

Figure 4.73: Runtimes

4.3.12 70x80x16

70x80x16	Moves			
	Best	*Worst*	*Average*	*Std.dev*
DigFill	25561	26683	26205	16.40
Real	46620	46772	46696	-
Greedy	38739	39484	39200	12.41
Greedy Dig	20869	21073	20941	6.48

	Manhattan			
	Best	*Worst*	*Average*	*Std. dev*
DigFill	1080862	1162332	1132100	133.82
Real	341470	340086	340778	-
Greedy	101932	115770	109138	50.11
Greedy Dig	288356	441189	360884	174.53

	Runtimes
	Iteration (ms)
DigFill	30121
Greedy	73659
Greedy Dig	18207

Figure 4.74: Table with moves, Manhattan distance and runtime in milliseconds

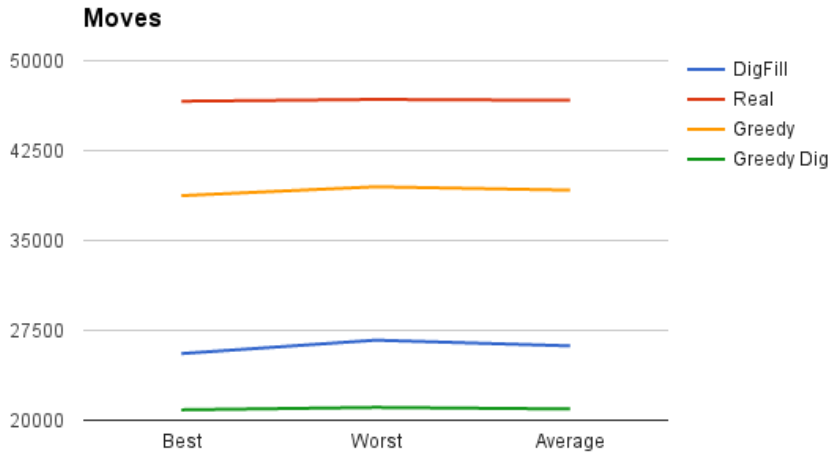

Figure 4.75: Best, worst and average moves

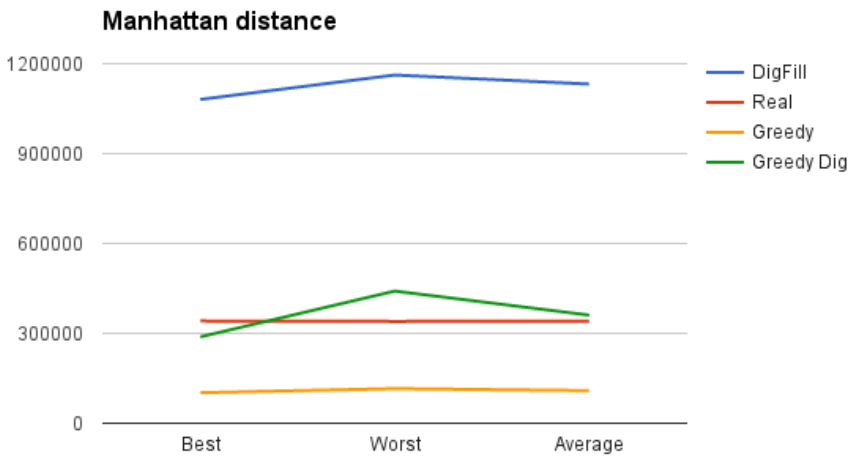

Figure 4.76: Best worst and average Manhattan distance

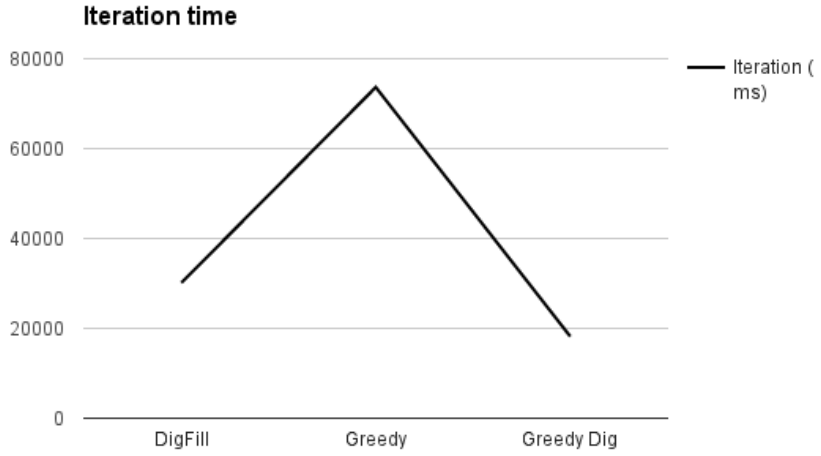

Figure 4.77: Runtimes

4.3.13 Move statistics

Table 4.9 shows how many objects are moved and how many times each object moves object is moved in each grid.

* assume lowest value of the three other values

20x20x10	Total objects	Total pops	Popped at least once	% popped at least once	Avg. times popped
DigFill	3640	3198	2865	79%	1.12
Real	3640	5438	2858	79%	1.90
Greedy	3640	5360	2869	79%	1.87
Greedy Dig	3640	2971	2858	79%	1.04

20x40x10	Total objects	Total pops	Popped at least once	% popped at least once	Avg. times popped
DigFill	7280	6249	5761	79%	1.08
Real	7280	11804	5760	79%	2.05
Greedy	7280	10766	5771	79%	1.87
Greedy Dig	7280	5934	5760	79%	1.03

20x40x16	Total objects	Total pops	Popped at least once	% popped at least once	Avg. times popped
DigFill	12080	10426	9701	80%	1.07
Real	12080	20154	9698	80%	2.08
Greedy	12080	18595	9780	81%	1.90
Greedy Dig	12080	9978	9698	80%	1.03

70x50x16	Total objects	Total pops	Popped at least once	% popped at least once	Avg. times popped
DigFill	24866	23668	18694	75%	1.27
Real	24866	41300	18693	75%	2.21
Greedy	24866	35178	18694	75%	1.88
Greedy Dig	24866	19065	18693	75%	1.02

70x80x16	Total objects	Total pops	Popped at least once	% popped at least once	Avg. times popped
DigFill	26522	25561	20363	77%	1.26
Real	26522	46620	20363	77%	2.29
Greedy	26522	38739	20373	77%	1.90
Greedy Dig	26522	20869	20363	77%	1.02

Table 4.9: Popped objects statistics

4.3.14 Summary

The standard Greedy DigFill algorithm gives a small improvement in number of moves compared to the actual simulation and the Manhattan distance is about a third. Manhattan distance per move is 2-3 for the greedy algorithm, while the actual simulations has a Manhattan distance per move between 6 and 9. As mentioned, these numbers are unsuitable for comparison and they are dependent on the total empty run distance in the actual system. If this distance is low, then the results from the greedy algorithm is very good. If the empty run distance is high, then the results might not be good at all.

When the fill step is removed, the Greedy Dig is almost optimal in terms of moves. The number of moves are reduced even more and the number of pops per popped object is now almost 1.0 for Greedy Dig. Hence, the objects are popped approximately 2-4% more times than the number of popped bins.

Pops per popped ratio now decreases towards an optimum, and it is a challenge to keep the Manhattan distance low at the same time. The average Manhattan

distance is actually a bit higher for the Greedy Dig algorithm than the results from the actual system, but with a high standard deviation the best result is far better than the average. The average Manhattan distance per move is much higher (from 7 to 18 dependent on grid), but that is a direct consequence of reducing number of moves. As mentioned, there is a clear trade off between these two and in order to reduce number of moves, we must move the objects longer to push them directly onto valid stacks.

Another positive effect of removing the fill step is that the complexity of the algorithm is reduced and it runs much faster. This is simply because the linear search (possibly 3n) for a fill stack is removed. Hence, more iterations can be run over a shorter period of time and increase the probability of finding even better solutions.

As it appears for now, Greedy DigFill optimizes Manhattan distance and the Greedy Dig optimizes number of moves. It should, however be possible for these two to meet at an optimal point in between. Where this optimal point is depends on many factors that are interrelated in a very complex way. By feeding an actual system with the output from both of these algorithms, the total reorganization time can be measured and compared to how the system performs today. Before this can be done, dependencies must be added for each move in the sequential list, and even more important; the list with dependencies must be parallelizable. Next section shows whether a sequential list of moves can be parallelized and to what degree the total Manhattan distance can be reduced.

4.4 Parallelization

From the initial DigFill version 1 to the current best versions, the solutions have been improved a lot. Small adjustments to the implementations make them suitable to optimize number of moves, the total Manhattan distance or both. However, before these solutions can be useful we need to show that the sequential lists can be parallelized such that multiple robots can carry out moves simultaneously.

The results in this section are measured using pickup and delivery dependencies as explained in section 3.7.2. Different algorithms are compared by their total Manhattan time and the total parallel Manhattan time of their solutions. The four different algorithms that will be compared are standard DigFill, Dig (a version of DigFill that only digs), Greedy DigFill and Greedy Dig.

Note that the best and worst solutions are the parallel best and worst. The best and worst total distances in the table may therefore not be the best and worst total distances, but they belong to the parallel distances. If the total best is a greater number than total worst, it is not a mistake, it is because the parallel distances decides the ordering.

4.4.1 20x20x10

20x20x10	Total Manhattan Distance			
	Best	*Worst*	*Average*	*Std.dev*
DigFill	46977	47176	46760	32.85
Dig	38725	40954	39905	24.50
Greedy DF	9957	10905	10458	19.52
Greedy Dig	24635	25573	25402	24.02

	Parallel Manhattan Distance			
	Best	*Worst*	*Average*	*Std. dev*
DigFill	17686	20548	19302	27.13
Dig	20385	22906	21633	20.35
Greedy DF	815	2084	1386	16.27
Greedy Dig	12443	14830	13721	22.49

	% of total		
	Best	*Worst*	*Average*
DigFill	37.65%	43.56%	41.28%
Dig	52.64%	55.93%	54.21%
Greedy DF	8.19%	19.11%	13.25%
Greedy Dig	50.51%	57.99%	54.02%

Figure 4.78: Table with parallel distances, total distances and parallel as % of total distance.

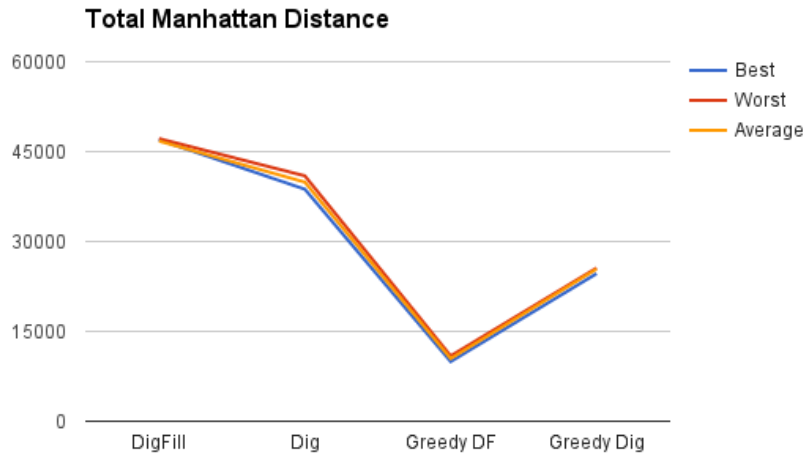

Figure 4.79: Best, worst and average parallel Manhattan distance.

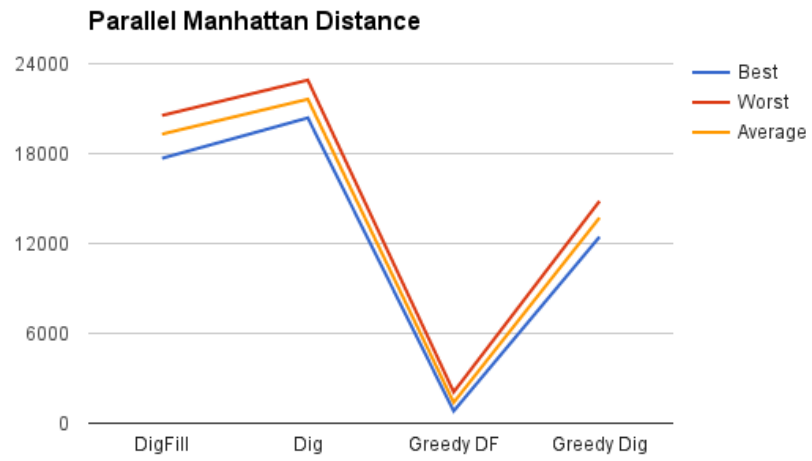

Figure 4.80: Best worst and average Manhattan distance.

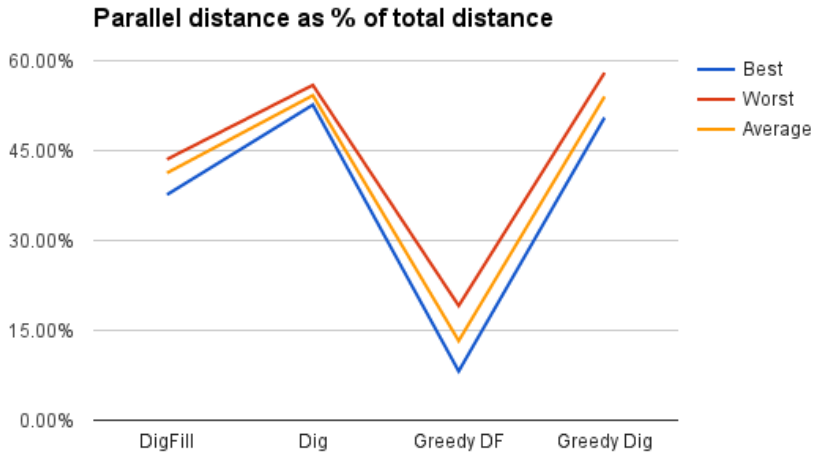

Figure 4.81: Parallel distance as % of total distance.

4.4.2 20x40x10

20x40x10	Total Manhattan Distance			
	Best	*Worst*	*Average*	*Std.dev*
DigFill	139717	134345	137080	66.6
Dig	118619	121100	119771	26.36
Greedy DF	21676	24297	23110	26.76
Greedy Dig	64319	70081	66996	30.01

	Parallel Manhattan Distance			
	Best	*Worst*	*Average*	*Std. dev*
DigFill	50326	57774	54255	41.45
Dig	63922	67357	65084	28.44
Greedy DF	1651	4443	3014	26.19
Greedy Dig	33120	40644	36503	43.41

	% of total		
	Best	*Worst*	*Average*
DigFill	36.02%	43.00%	39.58%
Dig	53.89%	55.62%	54.34%
Greedy DF	7.62%	18.29%	13.04%
Greedy Dig	51.49%	58.00%	54.49%

Figure 4.82: Table with parallel distances, total distances and parallel as % of total distance.

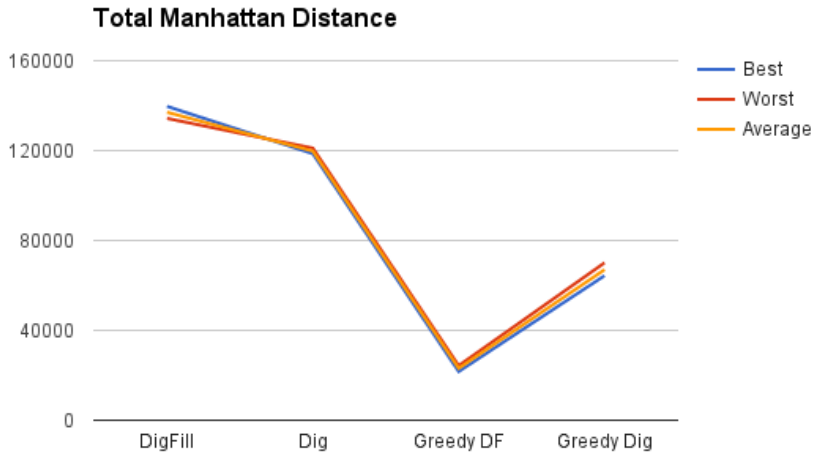

Figure 4.83: Best, worst and average parallel Manhattan distance.

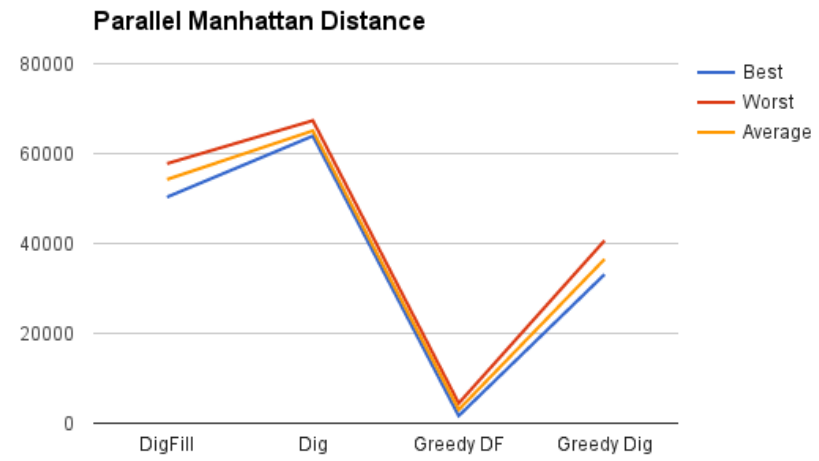

Figure 4.84: Best worst and average Manhattan distance.

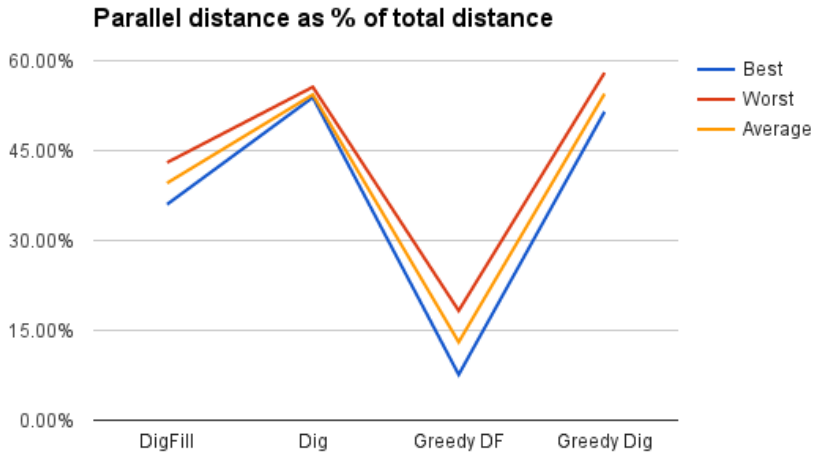

Figure 4.85: Parallel distance as % of total distance.

4.4.3 20x40x16

20x40x16	Total Manhattan Distance			
	Best	*Worst*	*Average*	*Std.dev*
DigFill	230095	220479	226882	89.29
Dig	195732	208604	202129	54.13
Greedy DF	41526	45113	43152	30.4
Greedy Dig	115094	133180	127863	58.39
	Parallel Manhattan Distance			
	Best	*Worst*	*Average*	*Std. dev*
DigFill	103648	116657	110080	56.02
Dig	125646	136120	130515	44.05
Greedy DF	3660	7773	5218	28.00
Greedy Dig	73566	99384	91512	66.03
	% of total			
	Best	*Worst*	*Average*	
DigFill	45.05%	52.91%	48.52%	
Dig	64.19%	65.25%	64.57%	
Greedy DF	8.81%	17.23%	12.09%	
Greedy Dig	63.92%	74.62%	71.57%	

Figure 4.86: Table with parallel distances, total distances and parallel as % of total distance.

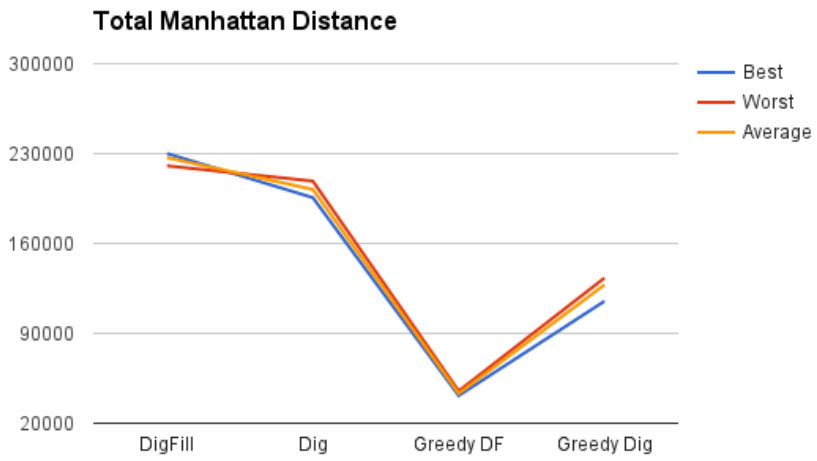

Figure 4.87: Best, worst and average parallel Manhattan distance.

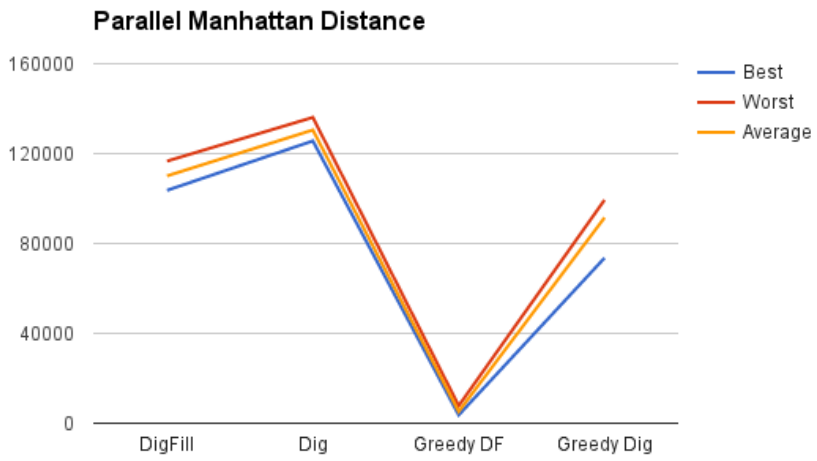

Figure 4.88: Best worst and average Manhattan distance.

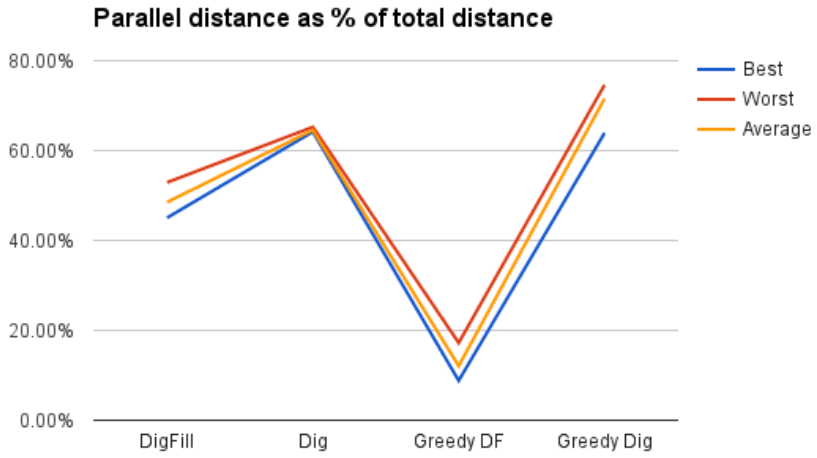

Figure 4.89: Parallel distance as % of total distance.

4.4.4 70x50x16

70x50x16	Total Manhattan Distance			
	Best	*Worst*	*Average*	*Std.dev*
DigFill	990453	1004798	992892	132.50
Dig	726390	749359	734112	73.36
Greedy DF	93420	106266	101115	47.7
Greedy Dig	259900	367402	323571	156.65

	Parallel Manhattan Distance			
	Best	*Worst*	*Average*	*Std. dev*
DigFill	414379	468205	447013	99.97
Dig	462021	480748	470411	65.20
Greedy DF	9719	14639	12452	35.03
Greedy Dig	100500	217142	168488	164.41

	% of total		
	Best	*Worst*	*Average*
DigFill	41.84%	46.60%	45.02%
Dig	63.61%	64.15%	64.08%
Greedy DF	10.40%	13.78%	12.31%
Greedy Dig	38.67%	59.10%	52.07%

Figure 4.90: Table with parallel distances, total distances and parallel as % of total distance.

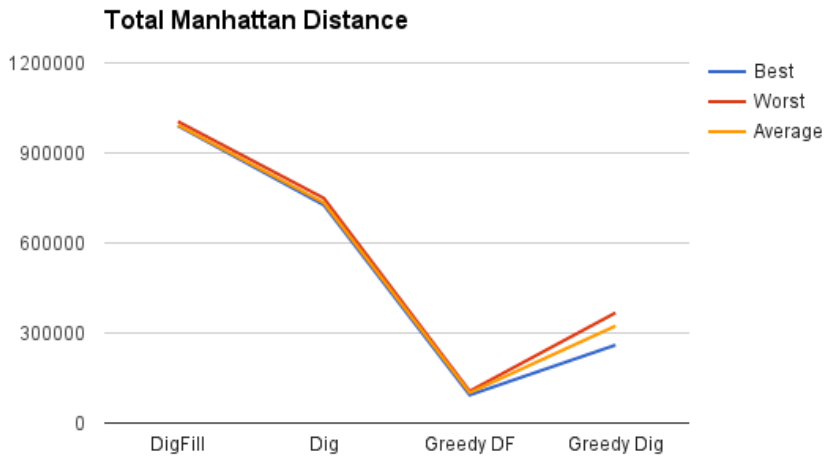

Figure 4.91: Best, worst and average parallel Manhattan distance.

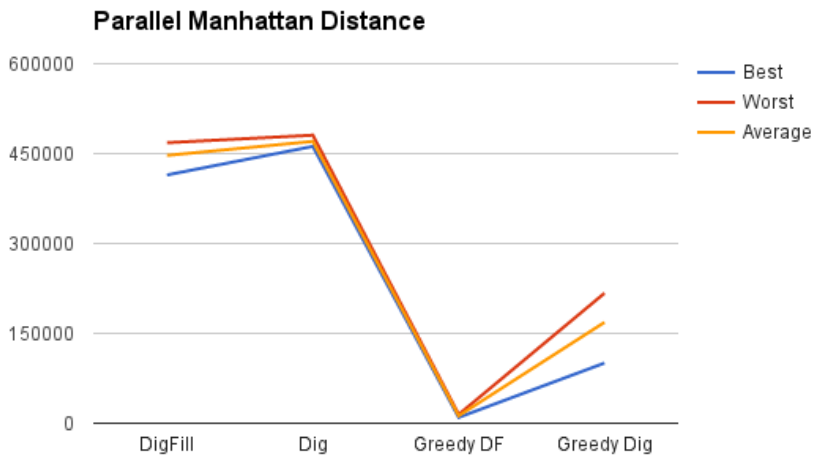

Figure 4.92: Best worst and average Manhattan distance.

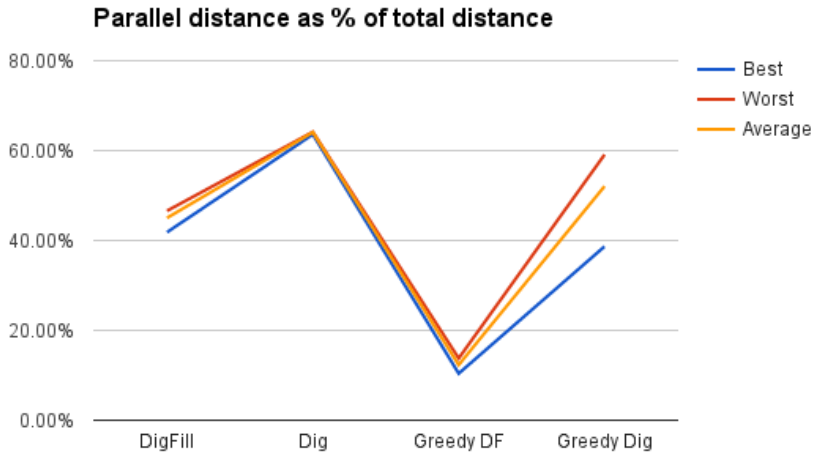

Figure 4.93: Parallel distance as % of total distance.

4.4.5 70x80x16

70x80x16	Total Manhattan Distance			
	Best	*Worst*	*Average*	*Std.dev*
DigFill	1115106	1154141	1130762	125.21
Dig	839363	860347	848757	73.28
Greedy DF	108056	113000	108415	49.38
Greedy Dig	283630	439199	356164	214.47

	Parallel Manhattan Distance			
	Best	*Worst*	*Average*	*Std. dev*
DigFill	482792	537416	509764	96.95
Dig	537379	556956	548254	65.02
Greedy DF	9498	15496	11587	36.00
Greedy Dig	105332	270112	183333	221.28

	% of total		
	Best	*Worst*	*Average*
DigFill	43.30%	46.56%	45.08%
Dig	64.02%	64.74%	64.59%
Greedy DF	8.79%	13.71%	10.69%
Greedy Dig	37.14%	61.50%	51.47%

Figure 4.94: Table with parallel distances, total distances and parallel as % of total distance.

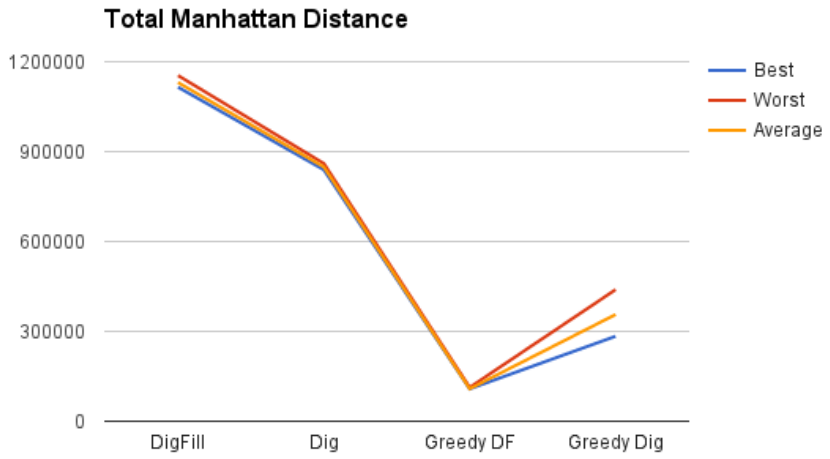

Figure 4.95: Best, worst and average parallel Manhattan distance.

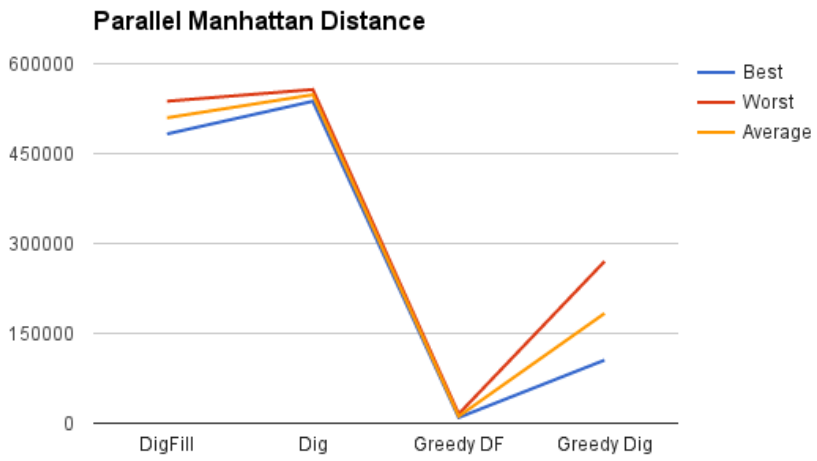

Figure 4.96: Best worst and average Manhattan distance.

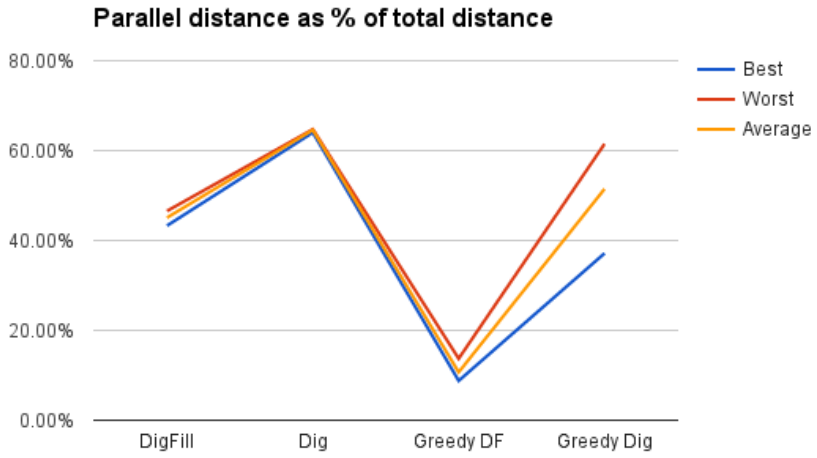

Figure 4.97: Parallel distance as % of total distance.

4.4.6 Summary

These results show the clear difference between an algorithm that allow the robots to work locally and an algorithm that use the entire grid to solve the problem. As expected, Greedy DigFill operates in local areas such that each dig and fill cycle can be independent of the other cycles. Thus, the total parallel Manhattan

distance around 10% of the total Manhattan distance.

The other three algorithms seem to be using much larger areas, and as a consequence the parallel Manhattan distance is larger when compared to the total Manhattan distance. However, the calculation of parallel Manhattan distance does not take empty runs and overhead into account and is probably very inaccurate when it comes to a realistic setting. To compensate for this issue, costs per move can be added to represent the distance of empty runs and other overhead.

Of course, how to set this cost correctly is also challenging. The overhead of a move can be things like pickup and delivery depth, number of turns, wait time for other robots, etc., so it is difficult to predict what this cost should be exactly. Because each move have some overhead, it is fair to assume that the cost is closely related to each move. Then, the equations below show a possible way to estimate the actual distance based on the distance from the simulations.

$$TotalEstimatedDistance = TotalDistance + cost \times moves$$

$$ParallelEstimatedDistance = ParallelDistance + cost \times moves$$

With these equations it is possible to calculate what the cost value must be if we want the estimated distances of two outputs to be equal to each other. Or, from a different perspective; if we know the cost C in an actual system, which algorithm should we choose? The table below shows the estimated cost and parallel Manhattan distances for the different grids. Number of moves used is the average number of moves from section 4.3. Calculations can also be done for total Manhattan distance, but it was not considered relevant at this point.

20x20x10					
Cost:	5.03				
	Total	*Parallel*	*Moves*	*Est.Total*	*Est.Parallel*
Greedy DF	10458	1386	5476	37994.2658	28922.2658
Greedy Dig	25402	13721	3023	40603.2658	28922.2658

20x40x10					
Cost:	6.60				
	Total	*Parallel*	*Moves*	*Est.Total*	*Est.Parallel*
Greedy DF	23110	3014	11085	78851.32695	76190.7327
Greedy Dig	66996	36503	6012	97227.56135	76190.7327

20x40x16					
Cost:	9.56				
	Total	*Parallel*	*Moves*	*Est.Total*	*Est.Parallel*
Greedy DF	43152	5218	19092	139156.8186	187769.2519
Greedy Dig	127863	91512	10067	178485.2768	187769.2519

70x50x16					
Cost:	9.55				
	Total	*Parallel*	*Moves*	*Est.Total*	*Est.Parallel*
Greedy DF	101115	12452	35500	279628.0453	351473.8496
Greedy Dig	323571	168488	19161	419922.7876	351473.8496

70x80x16					
Cost:	9.41				
	Total	*Parallel*	*Moves*	*Est.Total*	*Est.Parallel*
Greedy DF	108415	11587	39200	305533.6302	380306.1632
Greedy Dig	356164	183333	20941	461466.5826	380306.1632

Table 4.10: What is the overhead cost of the est. parallel distances are equal?

It is a clear pattern that as the grid size gets larger, the cost must be larger for Greedy Dig to be our choice of algorithm. Without knowing the cost, we can

say that (in general) Greedy DigFill with local moves is the best option for larger grids.

Because the maximum working capacity on the grid is decided by the number of robots, the total parallel distance is likely to be much larger when organizing an actual system. The numbers calculated in this section can be used to show that the sequential solutions are parallelizable and to consider which algorithm to use. The algorithms must be tested in an actual grid where tasks are delegated to a finite set of robots to see how they actually perform.

Chapter 5

Conclusion and further work

Section 1.2 first introduced the goal of the thesis which was to investigate different methods to reorganize the high density storage. The methods that have been discussed are ranging from entirely deterministic algorithms like the exhaustive search in section 3.2.2, to the totally random algorithms in section 3.3. We have seen that the algorithms that reside in between deterministic and random have provided the most promising results in this thesis.

5.1 Conclusion

5.1.1 Recap

In this thesis, the three most promising algorithms are DigFill (version 3), Greedy DigFill and Greedy Dig. Although the first one is considered the base algorithm, and the last two are using the base algorithm to optimize the solution in terms of Manhattan distance, I will include all three in the conclusion for comparison reasons.

The standard DigFill algorithm using GridConfiguration to decide valid moves, showed that it could return a solution that had fewer moves (best, worst and average) than today's system. At this point, the Manhattan distances were quite high, but for small grids with relatively short moves on average, choosing nearby stacks might not be that important. Because overhead of moves are considered to be significant, and there exist some reduction in moves that is equivalent to the increase in distance, there is a chance that even a standard DigFill could outperform today's system on small grids.

When the greedy search was introduced in Greedy DigFill, we could immediately see that the Manhattan distance was significantly reduced. The Manhattan distances were about a third compared to those from the actual simulation, but

with no empty runs this was as expected. Number of moves from the Greedy DigFill solutions are also lower than from what we would get from the actual simulation which reduces the total overhead of a reorganization. However, number of moves from Greedy DigFill are much higher than the moves from DigFill. It was discovered that the algorithm follows a pattern where it works very locally and the popped objects are likely to be pushed back onto the same stack in correct order. This discovery, the move-distance trade off, was discussed in section 4.3.7.

With pops per popped object close to 1.0 for the standard DigFill, but just below 2.0 for Greedy DigFill, Greedy Dig was implemented. Due to the improvement in DigFill version 3, it was now possible to drop the whole fill-step. Greedy Dig shows a pops per popped ratio even closer to 1.0 than what the initial DigFill could do, but due to the move-distance trade off, we would see an increased total distance.

Because these algorithms are based on some randomness, it is important to mention that only the best solution over multiple iterations will be chosen as the one set of moves to reorganize the storage. Because multiple solutions can be found and compared in seconds, it is in fact only the best solution that matters. Especially for Greedy Dig on larger grids, this is an important factor because the best Manhattan distance is far better than the average.

The ability to use these solutions such that multiple robots can work concurrently is also utterly important, and we see that Greedy DigFill works so locally that the minimum parallel Manhattan distance is about 10% of the total Manhattan distance. The other algorithms get a minimum parallel Manhattan distance around 40-50%. How parallelizable the sequential list needs to be would depend on number of robots and was not discussed in this thesis. The key point of this analysis was to confirm that the sequential lists can be divided into parallel moves.

5.1.2 Conclusion

Although, this thesis have had a theoretical focus and the results cannot be compared directly to how the actual system performs, I strongly believe that the algorithms presented here will reorganize the target storage faster than how the current system performs today. Between DigFill and Greedy Dig, the latter is strongly Pareto optimal, which leaves us with two algorithms; Greedy DigFill and Greedy Dig.

Table 4.9 in section 4.3.13 is probably the best indicator that the algorithms presented will show an improvement in total reorganization time when tested on on an actual system. From the table we can see that the number of moves is reduced, thus the move overhead is smaller. If we assume that the move distances in Greedy DigFill and Greedy Dig are more or less the same as the move distances in today's system, then these two algorithms should perform better.

Whether one of the algorithms (Greedy DigFill or Greedy Dig) is significantly better than the other, or if the optimal algorithm is a combination of these two, remains unknown for now. In terms of moves, there is no doubt that all algorithms presented will perform better than today's system. The overall impression is also that Greedy DigFill and Greedy Dig will perform better in terms of total reorganization time, compared to how the system performs today.

5.2 Further work

During my work on this thesis, I came up with new ideas every day. Unfortunately I did only have time to implement, discuss and test a few of them. This section provides an overview of my thoughts on small adjustment to the already implemented algorithms as well as new ideas that will require more implementation and a redesign of the algorithms used.

5.2.1 Compare and adjust

Due to lack of time and because I wanted to focus on the theory, outputs from the algorithms were never fed into the actual simulation system to compare the results properly. This should be done in order to see if the algorithms suggested here actually perform better when reorganizing the storage.

With data from simulations on the actual system, it is possible to adjust the algorithm to an optimal relationship between moves, distances, winding/straight moves, pickup and delivery depths, etc. For instance if deep pickups and deliveries takes a lot of time, then make sure the algorithm tries to avoid it as far as possible. If the straight moves are far better than winding moves, then choose the straight paths even though pickup and delivery stacks are further away from each other than the alternatives.

Algorithm runtime and storage reorganization

As expected, it is faster for a computer program running an algorithm to find the moves, than it is to complete all moves in a physical grid with moving robots. However, finding an optimal set of moves may be computationally heavy and take a while. Especially when working with large grids, finding good solutions can take time.

Because this is a problem with two steps (the vital step that finds a set of moves and the actual step that carries the moves out in the actual grid) we do not want the latter one to wait too long for the first to complete. If we allow the physical grid do at least some moves while the algorithm is searching for a final

solution, both systems can work simultaneously. When serving the robots some moves at an early stage, the algorithm get a lot of time to continue calculating while the robots are busy. When the robots are out of moves, the algorithm will provide them with even more moves and continue working.

Optimal number of robots

Because the algorithms are not run on an actual simulation system, we do not know whether parallelization of the sequential outputs from the algorithms is decent or not. We want all robots to be as busy as possible, but the main goal is to complete the reorganization as fast as possible. If an output that takes less advantage of robot capacity completes the reorganization faster than one with many busy robots, then the grid has too many robots. Fewer robots lead to more space for the robots left, and they can move more freely.

5.2.2 Choice of digStack, unload and fill stack

Different sortings of the digStack was tested, but results show that none of the particular sortings were much better than the others. Random selection was used in this thesis, but there might exist an optimal sorting.

A suggestion could be to choose neighboring stacks because there is probably a greater diversity of fill objects in an area close to a stack that has just been dug up. However, if stacks were always chosen close to the previous digStacks, we are prone to sequential dependencies. For instance if the algorithm start in one corner of the grid and moves row by row until the grid is solved, then the dependency graph is likely to have an intact path from the first to the last move. This issue speaks in favor of the random selection.

Another option is to make sure that dug up object do not block for the fill step. For instance if we are digging a stack such that the bottom high prioritized objects end up blocking all the non-prioritized objects on its neighbors, then fill objects must be picked up further away.

It should also be considered to not perform moves over long distances. For instance, when digging a stack and a suitable unload stack is found in the other end of the grid, unload the object nearby instead. This would be inconsistent with the base algorithm that always completes the dig-steps, but it might have a great impact on total Manhattan distance.

5.2.3 Reduce complexity of greedy search

As mentioned in 3.6 the neighbor list implementation that require a linear search is not an optimal way to implement the greedy search. Lists are simple to implement

and understand, but if these algorithms should run on larger grids than what we have used here, a far cheaper implementation is needed. In addition to using a huge amount of memory, linear time is not *that* fast.

By implementing a two dimensional segment tree, this operation can be done in logarithmic time instead of linear time. Briefly explained, a two dimensional segment tree is a data structure that divides the grid into nodes in a tree. Each stack is then a leaf node, and the stacks parent is a cluster of neighboring nodes. Of course, each stack has different neighbors, thus the two dimensions. The root is the entire grid. Each cluster as well as each stack has a variable that tells whether or not there is an object of a certain priority on top of this particular cluster/stack.

A search from stack S for a nearest neighbor N that needs the object X from S can be performed by walking up the tree, and when a cluster can confirm that it contains an N, then find the nearest N in that cluster. When the object is popped and pushed, the tree is updated.

A simpler variant that could improve the current implementation using lists, is to simply use some support structure to the neighbor lists. This structure can tell whether an N exists in some parts of the list or if it exists at all. If the N that needs X does not exist at all, then unnecessary neighbor searches can be avoided.

5.2.4 Evolutionary Strategies

The algorithms suggested in this thesis have been iterative, and all runs have been completed with solutions returned from every iteration. All of these solutions, except one, will be stashed such that we can keep the best one and use it to solve the reorganization problem. For n iterations, we have only one iteration that would be selected as our final solution, and the remaining $n - 1$ were only used to dilute the randomness. Now, what if these solutions could be pruned off and the iterations stopped at some earlier stage?

During the work with this thesis it was implemented an evolutionary strategy (ES) that works with a population of grids instead of one grid in each iteration. The *mutation* step is a fixed number of Greedy DigFill cycles, say 2, 4 or 8. Parent and survivor selection are based on a *fitness function*. The tested fitness function is the number of solved stacks in the individual grid, but a fitness function based on how many moves that are performed relative to how close the individual is to a solution is another possibility. However, when using the latter function it might be hard to balance the relationship between moves made and estimated moves left.

Because the fitness function is based on moves, while the mutation step on the Greedy DigFill algorithm, solutions returned from the ES had fewer moves than the iterative Greedy DigFill, but slightly higher Manhattan distance. The Manhattan distances were still much lower than the ones from Greedy Dig or standard DigFill.

Unfortunately, there was not time to do extensive studies on this methodology, but I do recommend that ES is further investigated in the future.

5.2.5 Divide and conquer

As the grids get larger, they take longer time to solve. This is simply because the set of stacks and objects are larger, thus the search space is also larger. However, the runtime increases far more than the increase in number of stacks and objects. Because the grid size is usually the same for all grids, while the area is expanded, it could be beneficial to split large grids into smaller zones and merge them as they get solved.

This would also help on the parallelization challenge. If a grid is divided into zones, then moves in separate zones cannot be dependent on each other before the two zones are merged. If robots also were assigned specific zones, this method would ensure that robots do not set out on long-haul trips.

There is however a downside with this implementation, and that is how the zones should be merged. Assume two zones have very different distribution of L1, L2 and non-prioritized objects. Then a merge would probably mean that valid objects must be dug up and moves again. On the other hand, if the zones have more or less the same distribution of different prioritized objects, then the merge would require less moves.

5.2.6 An (almost) complete DigFill search

Although the exhaustive search implemented and discussed in section 3.2.2 cannot be used to solve large grids, the implementation of DigFill (or DigFill only using the dig-step) and the corresponding results have opened for some new ideas. When it was discovered that the dig-step in DigFill was in fact redundant (if considering number of moves), the pops per popped object ratio went down to almost 1.0. Because there is only a finite set of objects that must be moves, then this is probably very close to an optimal solution. Clearly, 1.0 would be optimal, but it might not be possible for every grid.

Assume we have a grid with S stacks of max capacity C and the objects are more or less distributed such that is does not matter which digStack is chosen first digStack, i.e. digStacks can be chosen randomly. If the grid have a *1.0 solution*, then each digStack needs to be selected once and only once (because objects would never have to be moved again). For each object to be popped in digStack there are now $S - 1$ stacks to push the object to. This gives us now a complexity of $O(S \times C \times (S - 1))$, which is probably going to take a long time for large grids. However, some stacks will be full, but even more important more and more stacks

get solved. This means that the $S-1$ factor will be reduced for every cycle. Thus the average complexity is much lower.

This method is not implemented, and it is simply a concept that needs to be tested. The assumption that digStack selection does not matter makes the search not fully complete, in fact it is not even near complete if it was not based on DigFill (then we would have a situation where a stack can be popped once before moving on to another stack). The "one stack at a time" concept makes it less complex. Such an implementation would also be biased towards reducing number of moves, and it is not even nearly complete when it comes to Manhattan distance.

Such a search might turn out to be too complex, and we simply cannot deal with worst case scenarios in all types of problems. Overall, the results have been promising and they can very well be a good replacement for today's working algorithm. Theories in this last section should also be studied further to investigate if they can improve the algorithms to some extent.

Bibliography

[1] Gordana Dodig-Crnkovic. "Scientific methods in computer science". In: *Proceedings of the Conference for the Promotion of Research in IT at New Universities and at University Colleges in Sweden, Skövde, Suecia*. 2002, pp. 126–130.

[2] IFR Press Release. *Global Robotics Survey Industrial Robot Sales go through the roof Exceeding demand from Asia*. 2015. URL: http://www.ifr.org/news/ifr-press-release/global-robotics-survey-703/ (visited on 03/23/2015).

[3] IFR Press Release. *Service Robots Logistics systems on the rise*. 2014. URL: http://www.ifr.org/news/ifr-press-release/service-robots-logistic-systems-on-the-rise-657/ (visited on 03/23/2015).

[4] Kevin Kelly. "Better than human: Why robots will—and must—take our jobs". In: *Wired. http://www.wired.com/2012/12/ff-robots-will-take-our-jobs/(Accessed 4 August 2014.)* (2012).

[5] Paul E. Black. *Dictionary of Algorithms and Data Structures*. URL: http://www.nist.gov/dads/HTML/greedyalgo.html (visited on 03/27/2015).

[6] Richard D Alba. "A graph-theoretic definition of a sociometric clique†". In: *Journal of Mathematical Sociology* 3.1 (1973), pp. 113–126.

[7] Dino Karabeg and Rune Djurhuus. *Kompenium nr. 51 til IN210*. 1999, pp. 314–315.

[8] John Carl Villanueva. *How Many Atoms Are There in the Universe?* 2009. URL: http://www.universetoday.com/36302/atoms-in-the-universe/ (visited on 03/27/2015).

[9] Roger Antonsen. *Logiske Metoder*. Universitetsforlaget, 2014. Chap. 18, pp. 203–203.